CS252
Graduate Computer Architecture
Lecture 20
April 12th, 2010

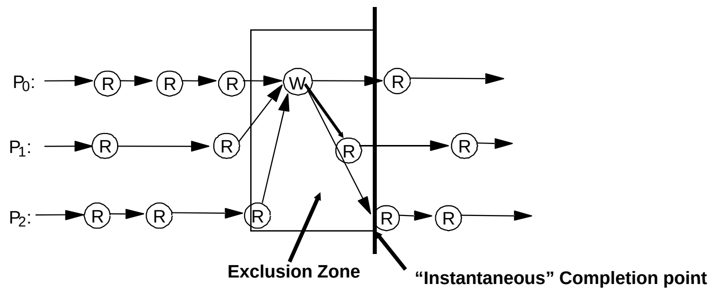
Distributed Shared Memory

Prof John D. Kubiatowicz
http://www.cs.berkeley.edu/~kubitron/cs252

Recall: Sequential Consistency Example

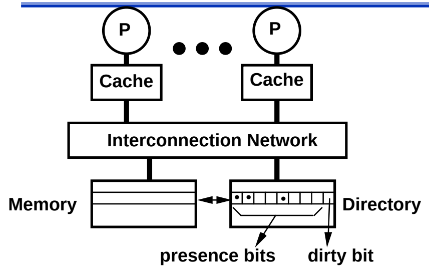
Processor 1	Processor 2	One Consistent Serial Order
LD_1 A ⇒ 5	LD_5 B ⇒ 2	LD_1 A ⇒ 5
LD_2 B ⇒ 7	…	LD_2 B ⇒ 7
ST_1 A,6	LD_6 A ⇒ 6	LD_5 B ⇒ 2
…	ST_4 B,21	ST_1 A,6
LD_3 A ⇒ 6	…	LD_6 A ⇒ 6
LD_4 B ⇒ 21	LD_7 A ⇒ 6	ST_4 B,21
ST_2 B,13	…	LD_3 A ⇒ 6
ST_3 B,4	LD_8 B ⇒ 4	LD_4 B ⇒ 21
		LD_7 A ⇒ 6
		ST_2 B,13
		ST_3 B,4
		LD_8 B ⇒ 4

Recall: Ordering: Scheurich and Dubois



- Sufficient Conditions
  - every process issues mem operations in program order
  - after a write operation is issued, the issuing process waits for the write to complete before issuing next memory operation
  - after a read is issued, the issuing process waits for the read to complete and for the write whose value is being returned to complete (gloabaly) before issuing its next operation

Basic Operation of Directory

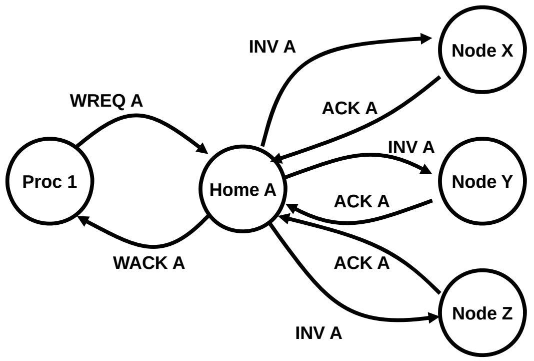


- k processors.
- With each cache-block in memory: k presence-bits, 1 dirty-bit
- With each cache-block in cache: 1 valid bit, and 1 dirty (owner) bit

- Read from main memory by processor i:
  - If dirty-bit OFF then { read from main memory; turn p[i] ON; }
  - If dirty-bit ON then { recall line from dirty proc (cache state to shared); update memory; turn dirty-bit OFF; turn p[i] ON; supply recalled data to i;}
- Write to main memory by processor i:
  - If dirty-bit OFF then {send invalidations to all caches that have the block; turn dirty-bit ON; supply data to i; turn p[i] ON; ... }
  - If dirty-bit ON then {recall line from dirty proc (invalidate); update memory; keep dirty-bit ON; supply recalled data to i}

Example: How to invalidate read copies

- Example: from read-shared to read-write



- Advantages: No need to broadcast to hunt down copies
  - Some entity in the system knows where all copies reside
  - Doesn't need to be specific – could reflect *group* of processors each of which might contain a copy (or might not)

Scaling Issues

- Memory and directory bandwidth
  - Centralized directory is bandwidth bottleneck, just like centralized memory
  - How to maintain directory information in distributed way?
- Performance characteristics
  - traffic: no. of network transactions each time protocol is invoked
  - latency = no. of network transactions in critical path
- Directory storage requirements
  - Number of presence bits grows as the number of processors
- Deadlock issues:
  - May need as many networks as longest chain of request/response pairs
- How directory is organized affects all these, performance at a target scale, as well as coherence management issues

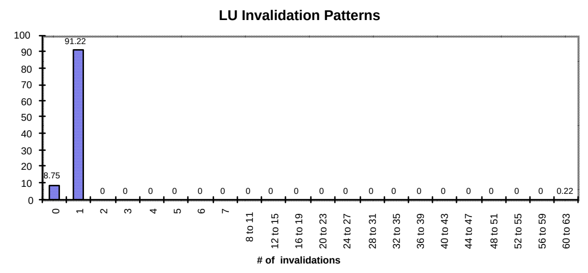
Insight into Directory Requirements

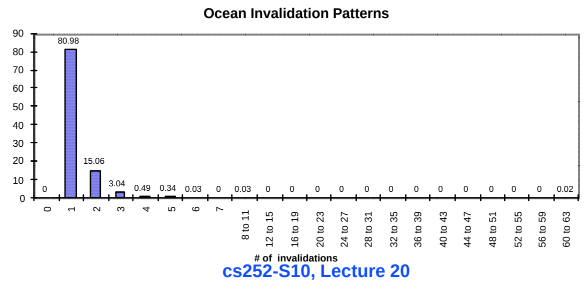
- If most misses involve O(P) transactions, might as well broadcast!

⇒ Study Inherent program characteristics:
  - frequency of write misses?
  - how many sharers on a write miss
  - how these scale

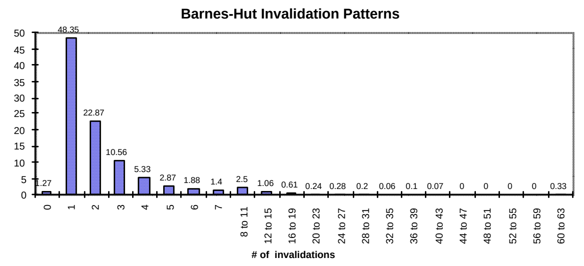
- Also provides insight into how to organize and store directory information

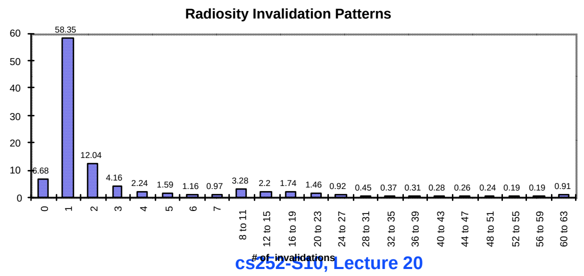
Cache Invalidation Patterns





Cache Invalidation Patterns

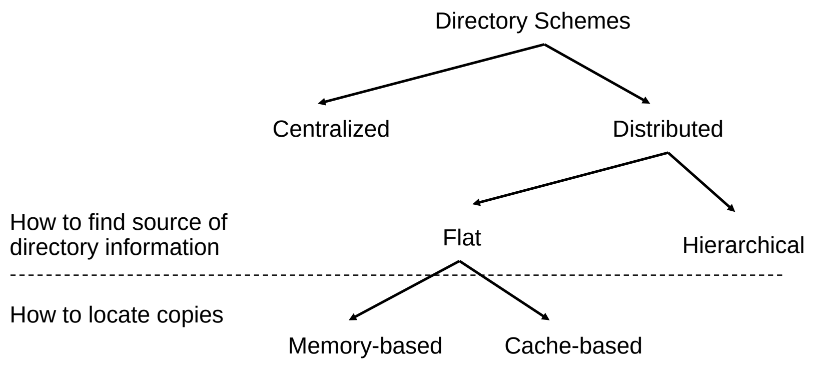




Sharing Patterns Summary

- Generally, few sharers at a write, scales slowly with P
  - Code and read-only objects (e.g, scene data in Raytrace)
    - » no problems as rarely written
  - Migratory objects (e.g., cost array cells in LocusRoute)
    - » even as # of PEs scale, only 1-2 invalidations
  - Mostly-read objects (e.g., root of tree in Barnes)
    - » invalidations are large but infrequent, so little impact on performance
  - Frequently read/written objects (e.g., task queues)
    - » invalidations usually remain small, though frequent
  - Synchronization objects
    - » low-contention locks result in small invalidations
    - » high-contention locks need special support (SW trees, queueing locks)
- Implies directories very useful in containing traffic
  - if organized properly, traffic and latency shouldn't scale too badly
- Suggests techniques to reduce storage overhead

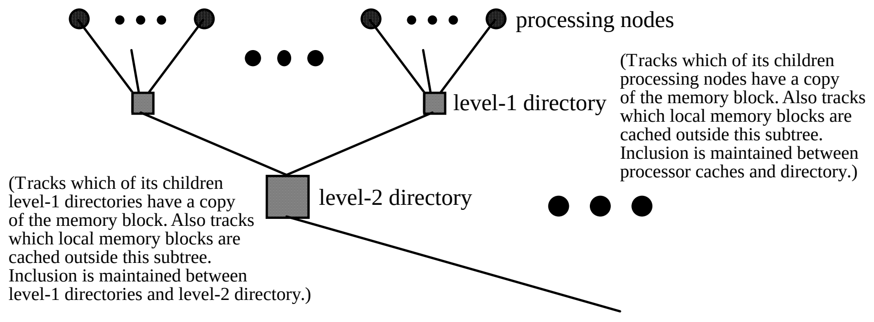
Organizing Directories



How to Find Directory Information

- centralized memory and directory - easy: go to it
  - but not scalable
- distributed memory and directory
  - flat schemes
    - » directory distributed with memory: at the home
    - » location based on address (hashing): network xaction sent directly to home
  - hierarchical schemes
    - » ??

How Hierarchical Directories Work


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 processor caches and directory.)
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-1 directories and level-2 directory.)

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

Find Directory Info (cont)

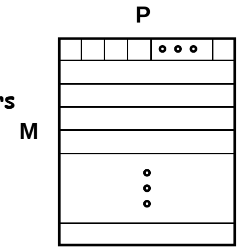
- distributed memory and directory
  - flat schemes
    - » hash
  - hierarchical schemes
    - » node's directory entry for a block says whether each subtree caches the block
    - » to find directory info, send "search" message up to parent
      - routes itself through directory lookups
    - » like hiearchical snooping, but point-to-point messages between children and parents

How Is Location of Copies Stored?

- Hierarchical Schemes
  - through the hierarchy
  - each directory has presence bits child subtrees and dirty bit
- Flat Schemes
  - vary a lot
  - different storage overheads and performance characteristics
  - Memory-based schemes
    - » info about copies stored all at the home with the memory block
    - » Dash, Alewife , SGI Origin, Flash
  - Cache-based schemes
    - » info about copies distributed among copies themselves
      - each copy points to next
    - » Scalable Coherent Interface (SCI: IEEE standard)

Flat, Memory-based Schemes

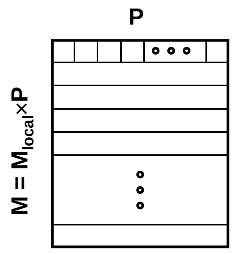
- info about copies co-located with block at the home
  - just like centralized scheme, except distributed
- Performance Scaling
  - traffic on a write: proportional to number of sharers
  - latency on write: can issue invalidations to sharers in parallel



- Storage overhead
  - simplest representation: *full bit vector*, (called "Full-Mapped Directory"), i.e. one presence bit per node
  - storage overhead doesn't scale well with P; 64-byte line implies
    - » 64 nodes: 12.7% ovhd.
    - » 256 nodes: 50% ovhd.; 1024 nodes: 200% ovhd.
  - for M memory blocks in memory, storage overhead is proportional to P*M:
    - » Assuming each node has memory M_{local}= M/P, $\propto P^2 M_{local}$
    - » This is why people talk about full-mapped directories as scaling with the square of the number of processors

Reducing Storage Overhead

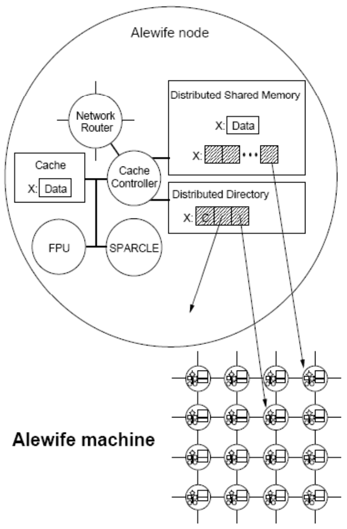
- Optimizations for full bit vector schemes
  - increase cache block size (reduces storage overhead proportionally)
  - use multiprocessor nodes (bit per mp node, not per processor)
  - still scales as P*M, but reasonable for all but very large machines
    - » 256-procs, 4 per cluster, 128B line: 6.25%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



Storage Reductions

- Width observation:
  - most blocks cached by only few nodes
  - don't have a bit per node, but entry contains a few pointers to sharing nodes
    - » Called "Limited Directory Protocols"
  - P=1024 => 10 bit ptrs, can use 100 pointers and still save space
  - sharing patterns indicate a few pointers should suffice (five or so)
  - need an overflow strategy when there are more sharers
- Height observation:
  - number of memory blocks >> number of cache blocks
  - most directory entries are useless at any given time
  - Could allocate directory from pot of directory entries
    - » If memory line doesn't have a directory, no-one has copy
    - » What to do if overflow? Invalidate directory with invaliations
  - organize directory as a cache, rather than having one entry per memory block

Case Study: Alewife Architecture



- Cost Effective Mesh Network
  - Pro: Scales in terms of hardware
  - Pro: Exploits Locality
- Directory Distributed along with main memory
  - Bandwidth scales with number of processors
- Con: Non-Uniform Latencies of Communication
  - Have to manage the mapping of processes/threads onto processors due
  - Alewife employs techniques for latency minimization and latency tolerance so programmer does not have to manage
- Context Switch in 11 cycles between processes on remote memory request which has to incur communication network latency
- Cache Controller holds tags and implements the coherence protocol

LimitLESS Protocol (Alewife)

- Limited Directory that is Locally Extended through Software Support
- Handle the common case (small worker set) in hardware and the exceptional case (overflow) in software
- Processor with rapid trap handling (executes trap code within 4 cycles of initiation)
- State Shared
  - Processor needs complete access to coherence related controller state in the hardware directories
  - Directory Controller can invoke processor trap handlers
- Machine needs an interface to the network that allows the processor to launch and intercept coherence protocol packets

The Protocol

Type	Symbol	Name	Data?
Cache to Memory	RREQ	Read Request	
	WREQ	Write Request	
	REPM	Replace Modified	√
	UPDATE	Update	√
	ACKC	Invalidate Ack.	
Memory to Cache	RDATA	Read Data	√
	WDATA	Write Data	√
	INV	Invalidate	
	BUSY	Busy Signal	

Table 1: Protocol messages for hardware coherence.

Component	Name	Meaning
Memory	Read-Only	Some number of caches have read-only copies of the data.
	Read-Write	Exactly one cache has a read-write copy of the data.
	Read-Transaction	Holding read request, update is in progress.
	Write-Transaction	Holding write request, invalidation is in progress.
Cache	Invalid	Cache block may not be read or written.
	Read-Only	Cache block may be read, but not written.
	Read-Write	Cache block may be read or written.

Table 2: Directory states.

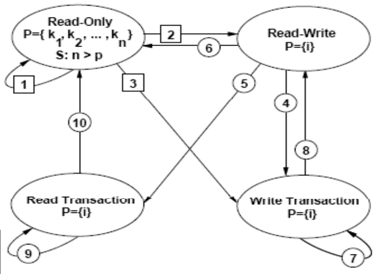
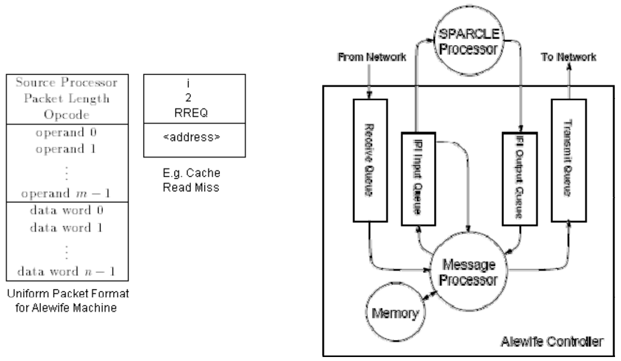


Figure 2: Directory state transition diagram.

- **Alewife: p=5-entry limited directory with software extension (LimitLESS)**
- **Read-only directory transaction:**
  - Incoming RREQ with n ≤ p ⇒ Hardware memory controller responds
  - If n > p: send RREQ to processor for handling

Transition to Software



- Trap routine can either discard packet or store it to memory
- Store-back capability permits message-passing and block transfers
- Potential Deadlock Scenario with Processor Stalled and waiting for a remote cache-fill
  - Solution: Synchronous Trap (stored in local memory) to empty input queue

Transition to Software (Con't)

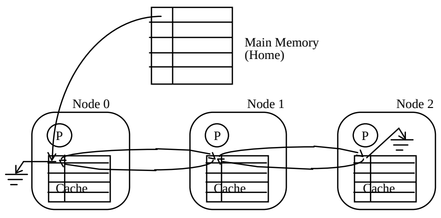
Meta State	Description
Normal	Directory handled by hardware.
Trans-In-Progress	Interlock, software processing.
Trap-On-Write	Trap: WREQ, UPDATE, REPM.
Trap-Always	Trap: all incoming packets.

Table 4: Directory meta states for LimitLESS scheme.

- Overflow Trap Scenario
  - First Instance: Full-Map bit-vector allocated in local memory and hardware pointers transferred into this and vector entered into hash table
  - Otherwise: Transfer hardware pointers into bit vector
  - Meta-State Set to "Trap-On-Write"
  - While emptying hardware pointers, Meta-State: "Trans-In-Progress"
- Incoming Write Request Scenario
  - Empty hardware pointers to memory
  - Set AckCtr to number of bits that are set in bit-vector
  - Send invalidations to all caches except possibly requesting one
  - Free vector in memory
  - Upon invalidate acknowledgement (AckCtr == 0), send Write-Permission and set Memory State to "Read-Write"

Flat, Cache-based Schemes

- How they work:
  - home only holds pointer to rest of directory info
  - distributed linked list of copies, weaves through caches
    - » cache tag has pointer, points to next cache with a copy
  - on read, add yourself to head of the list (comm. needed)
  - on write, propagate chain of invals down the list
- Scalable Coherent Interface (SCI) IEEE Standard
  - doubly linked list



Scaling Properties (Cache-based)

- Traffic on write: proportional to number of sharers
- Latency on write: proportional to number of sharers!
  - don't know identity of next sharer until reach current one
  - also assist processing at each node along the way
  - (even reads involve more than one other assist: home and first sharer on list)
- Storage overhead: quite good scaling along both axes
  - Only one head ptr per memory block
    - » rest is all prop to cache size
- Very complex!!!

Summary of Directory Organizations

- Flat Schemes:
- Issue (a): finding source of directory data
  - go to home, based on address
- Issue (b): finding out where the copies are
  - memory-based: all info is in directory at home
  - cache-based: home has pointer to first element of distributed linked list
- Issue (c): communicating with those copies
  - memory-based: point-to-point messages (perhaps coarser on overflow)
    - » can be multicast or overlapped
  - cache-based: part of point-to-point linked list traversal to find them
    - » serialized
- Hierarchical Schemes:
  - all three issues through sending messages up and down tree
  - no single explict list of sharers
  - only direct communication is between parents and children

Summary of Directory Approaches

- Directories offer scalable coherence on general networks
  - no need for broadcast media
- Many possibilities for organizing directory and managing protocols
- Hierarchical directories not used much
  - high latency, many network transactions, and bandwidth bottleneck at root
- Both memory-based and cache-based flat schemes are alive
  - for memory-based, full bit vector suffices for moderate scale
    - » measured in nodes visible to directory protocol, not processors
  - will examine case studies of each

Summary

- Memory Coherence:
  - Writes to a given location eventually propagated
  - Writes to a given location seen in same order by everyone
- Memory Consistency:
  - Constraints on ordering between processors and locations
- Sequential Consistency:
  - For every parallel execution, there exists a serial interleaving
- Distributed Directory Structure
  - Flat: Each address has a "home node"
  - Hierarchical: directory spread along tree
- Mechanism for locating copies of data
  - Memory-based schemes
    - » info about copies stored all at the home with the memory block
  - Cache-based schemes
    - » info about copies distributed among copies themselves