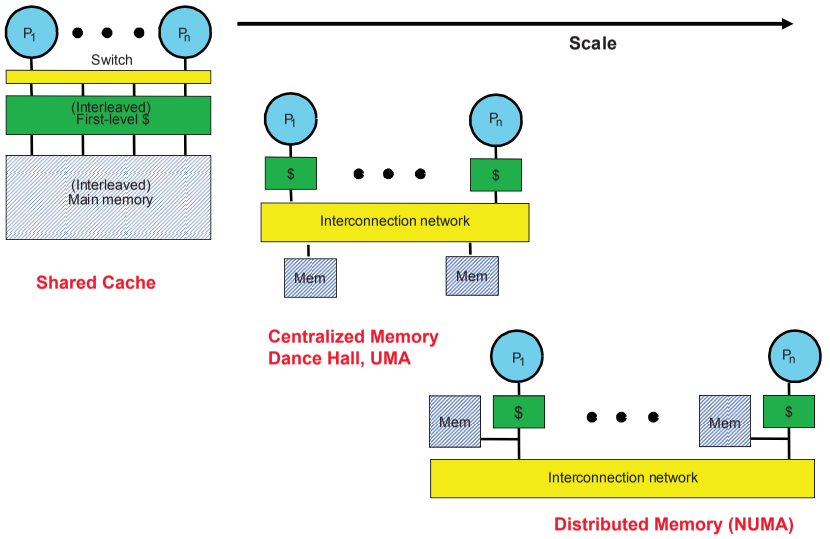
CS252
Graduate Computer Architecture
Lecture 19
April 4th, 2012

Bus-Based Shared Memory (Con't)
Distributed Shared Memory

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

Recall: Natural Extensions of Memory System



Recall: Definitions

- Memory operation
  - load, store, read-modify-write
- Issues
  - leaves processor's internal environment and is presented to the memory subsystem (caches, buffers, busses,dram, etc)
- Performed with respect to a processor
  - write: subsequent reads return the value
  - read: subsequent writes cannot affect the value
- Coherent Memory System
  - Writes to a given location eventually propagated
  - Writes to a given location seen in same order by everyone, i.e. there exists a serial order of mem operations on each location s.t.
    - » operations issued by a process appear in order issued
    - » value returned by each read is that written by previous write in the serial order

=> write propagation + write serialization

Recall: Is 2-state Protocol Coherent?

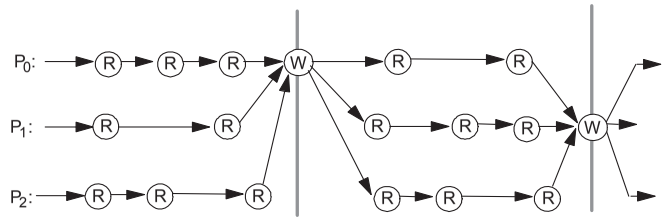
- Assume bus transactions and memory operations are atomic, one-level cache
  - all phases of one bus transaction complete before next one starts
  - processor waits for memory op to complete before issuing next
  - with one-level cache, assume invalidations applied during bus xaction
- All writes go to bus + atomicity
  - Writes serialized by order in which they appear on bus (bus order)

  ⇒ invalidations applied to caches in bus order
- How to insert reads in this order?
  - Important since processors see writes through reads, so determines whether write serialization is satisfied
  - But read hits may happen independently and do not appear on bus or enter directly in bus order

Recall: Ordering Reads in 2-state protocol

- Read misses
  - appear on bus, and will "see" last write in bus order
- Read hits: do not appear on bus
  - But value read was placed in cache by either
    - » most recent write by this processor, or
    - » most recent read miss by this processor
  - Both these transactions appeared on the bus
  - So reads hits also see values as produced bus order

Ordering for 2-state protocol



- Writes establish a partial order
- Doesn't constrain ordering of reads, though bus will order read misses too
  - any order among reads between writes is fine, as long as in program order

Setup for Mem. Consistency

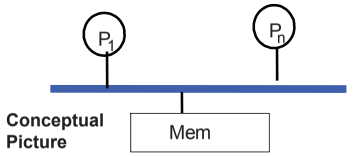
- Coherence ⇒ Writes to a location become visible to all in the same order
- But when does a write become visible?

- How do we establish orders between a write and a read by different procs?
  - use event synchronization
- Typically use more than one location!

Example

P_1	P_2
/*Assume initial value of A and flag is 0*/	
`A = 1;`	`while (flag == 0);` /*spin idly*/
`flag = 1;`	`print A;`

- Intuition not guaranteed by coherence
- expect memory to respect order between accesses to *different* locations issued by a given process
  - to preserve orders among accesses to same location by different processes
- Coherence is not enough!
  - pertains only to single location



Another Example of Ordering?

P_1	P_2
/*Assume initial values of A and B are 0 */	
(1a) A = 1;	(2a) print B;
(1b) B = 2;	(2b) print A;

- What's the intuition?
  - Whatever it is, we need an ordering model for clear semantics
    - » across different locations as well
    - » so programmers can reason about what results are possible
  - This is the memory consistency model

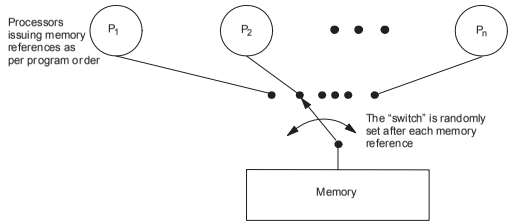
Memory Consistency Model

- Specifies constraints on the order in which memory operations (from any process) can appear to execute with respect to one another
  - What orders are preserved?
  - Given a load, constrains the possible values returned by it
- Without it, can't tell much about an SAS program's execution
- Implications for both programmer and system designer
  - Programmer uses to reason about correctness and possible results
  - System designer can use to constrain how much accesses can be reordered by compiler or hardware
- Contract between programmer and system

Sequential Consistency

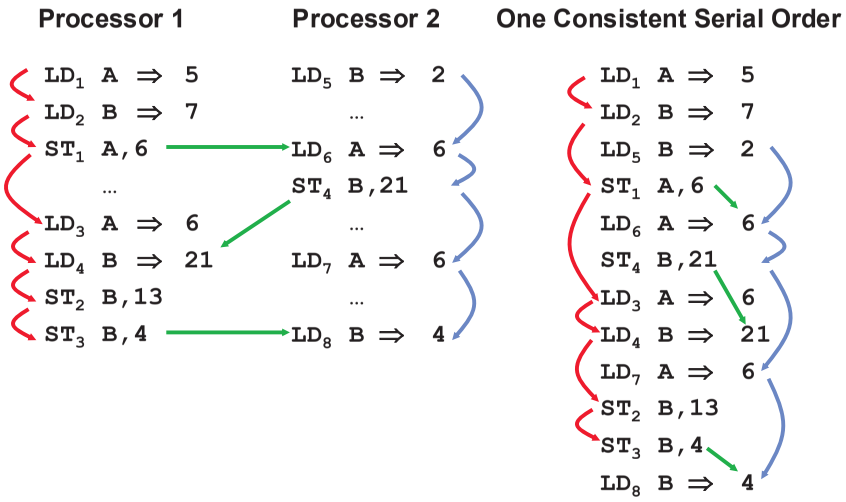
- Memory operations from a proc become visible (to itself and others) in program order
- There exists a total order, consistent with this partial order - i.e., an interleaving
  - the position at which a write occurs in the hypothetical total order should be the same with respect to all processors
- Said another way:
  - For any possible individual run of a program on multiple processors
  - Should be able to come up with a serial interleaving of all operations that respects
    - » Program Order
    - » Read-after-write orderings (locally and through network)
    - » Also Write-after-read, write-after-write

Sequential Consistency

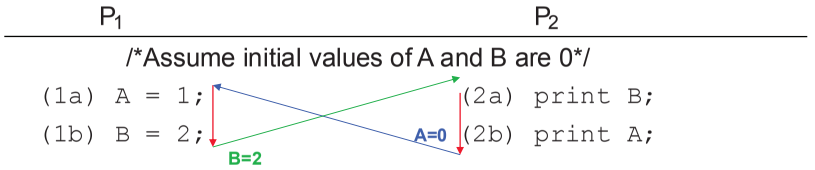


- Total order achieved by *interleaving* accesses from different processes
  - Maintains *program order*, and memory operations, from all processes, appear to [issue, execute, complete] atomically w.r.t. others
  - as if there were no caches, and a single memory
- "A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [Lamport, 1979]

Sequential Consistency Example



SC Example

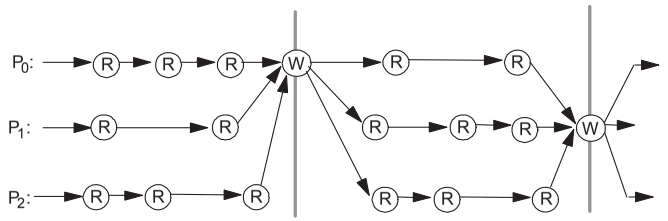


- What matters is order in which operations *appear to execute*, not the chronological order of events
- Possible outcomes for (A,B): (0,0), (1,0), (1,2)
- What about (0,2) ?
  - program order ⇒ 1a->1b and 2a->2b
  - A = 0 implies 2b->1a, which implies 2a->1b
  - B = 2 implies 1b->2a, which leads to a contradiction (cycle!)
- Since there is a cycle⇒no sequential order that is consistent!
  - Alternatively – no sequentially consistent machine could produce this result!

Implementing SC for general case

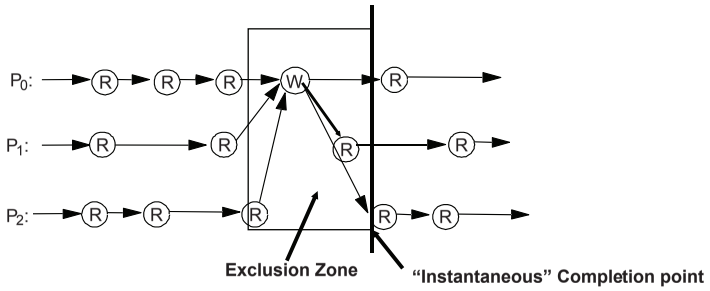
- Two kinds of requirements
  - Program order
    - » memory operations issued by a process must appear to execute (become visible to others and itself) in program order
  - Atomicity
    - » in the overall hypothetical total order, one memory operation should appear to complete with respect to all processes before the next one is issued
    - » guarantees that total order is consistent across processes
  - tricky part is making writes atomic
- How can compilers violate SC?
- How can architectural enhancements violate SC?

Happens Before: arrows are time



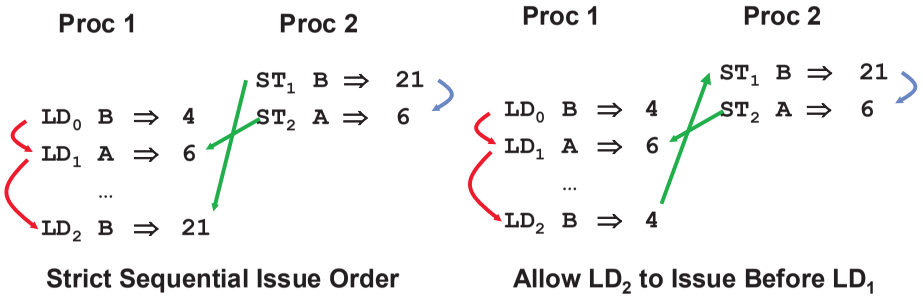
- Tricky part is relationship between nodes with respect to single location
  - Program order adds relationship between locations
- Easy topological sort comes up with sequential ordering assuming:
  - All happens-before relationships are time
  - Then – can't have time cycles (at least not inside classical machine in normal spacetime ☺).
- Unfortunately, writes are not instantaneous
  - What do we do?

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

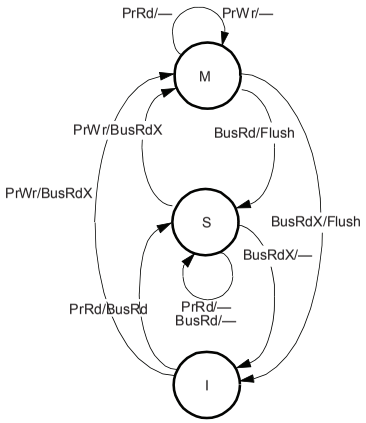
What about reordering of accesses?



- Can LD_2 issue before LD_1?
  - Danger of getting CYCLE! (i.e. not sequentially consistent)
- What can we do?
  - Go ahead and issue ld early, but watch cache
  - If value invalidated from cache early:
    - » Must squash LD_2 and any instructions that have used its value
- Reordering of Stores
  - Must be even more careful

MSI Invalidate Protocol: Write Back Cache

- Three States:
  - "M": "Modified"
  - "S": "Shared"
  - "I": "Invalid"
- Read obtains block in "shared"
  - even if only cache copy
- Obtain exclusive ownership before writing
  - BusRdx causes others to invalidate (demote)
  - If M in another cache, will flush
  - BusRdx even if hit in S
    - » promote to M (upgrade)
- What about replacement?
  - S->I, M->I as before



Write Serialization for Coherence

- Correctness
  - When is write miss performed?
    - » How does writer "observe" write?
    - » How is it "made visible" to others?
    - » How do they "observe" the write?
  - When is write hit made visible to others?
  - When does a write hit complete globally?
- Writes that appear on the bus (BusRdX) are ordered by bus
  - performed in writer's cache before other transactions, so ordered same w.r.t. all processors (incl. writer)
  - Read misses also ordered wrt these
- Write that don't appear on the bus:
  - P issues BusRdX B.
  - further mem operations on B until next transaction are from P
    - » read and write hits
    - » these are in program order
  - for read or write from another processor
    - » separated by intervening bus transaction
- Reads hits?

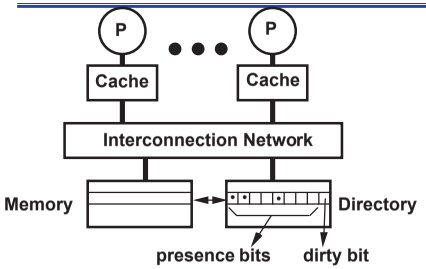
Sequential Consistency

- Bus imposes total order on xactions for all locations
- Between xactions, procs perform reads/writes (locally) in program order
- So any execution defines a natural partial order
  - M_j subsequent to M_i if
    - » (i) M_j follows M_i in program order on same processor,
    - » (ii) M_j generates bus xaction that follows the memory operation for M_i
- In segment between two bus transactions, any interleaving of local program orders leads to consistent total order
- Within segment writes observed by proc P serialized as:
  - Writes from other processors by the previous bus xaction P issued
  - Writes from P by program order
  - Insight: only one cache may have value in "M" state at a time…

Sufficient conditions

- Sufficient Conditions
  - issued in program order
  - after write issues, the issuing process waits for the write to complete before issuing next memory operation
  - after read is issues, the issuing process waits for the read to complete and for the write whose value is being returned to complete (globally) before issuing its next operation
- Write completion
  - can detect when write appears on bus (flush) appears
- Write atomicity:
  - if a read returns the value of a write, that write has become visible to all others already
    - » Either: it is being read by the processor that wrote it and no other processor has a copy (thus any read by any other processor will get new value via a flush
    - » Or: it has already been flushed back to memory and all processors will have the value

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}

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
- How directory is organized affects all these, performance at a target scale, as well as coherence management issues

Insight into Directory Requirements

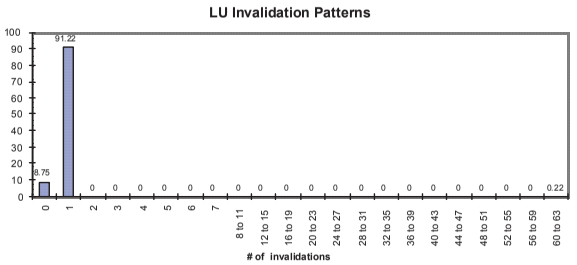
- If most misses involve O(P) transactions, might as well broadcast!

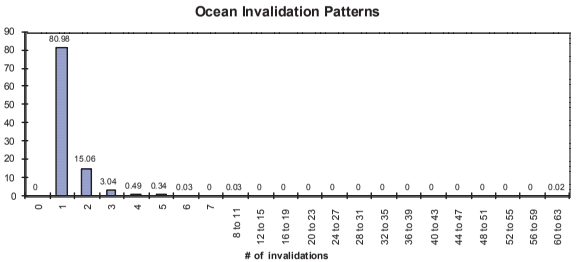
⇒ Study Inherent program characteristics:

  - frequency of write misses?
  - how many sharers on a write miss
  - how these scale

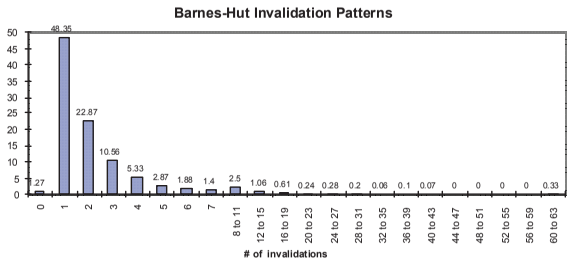
- Also provides insight into how to organize and store directory information

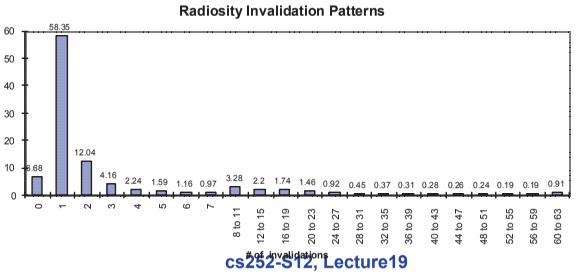
Cache Invalidation Patterns





Cache Invalidation Patterns

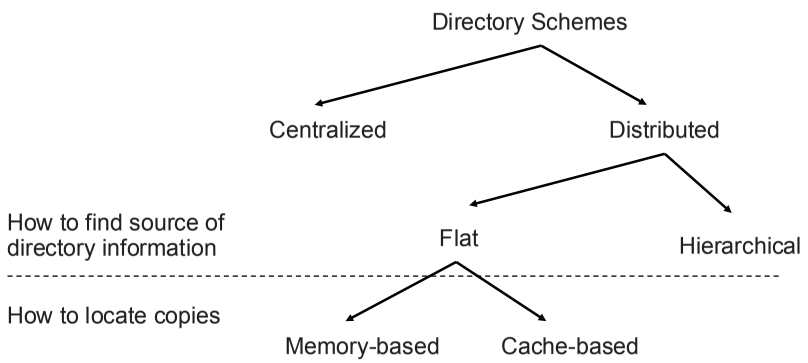




Sharing Patterns Summary

- 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

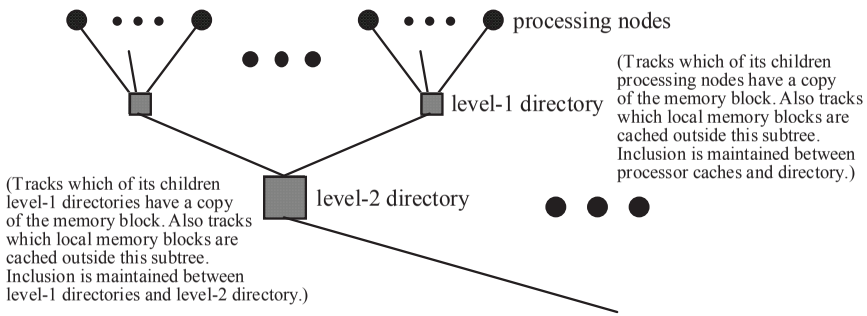
Organizing Directories



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



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

Find Directory Info (cont)

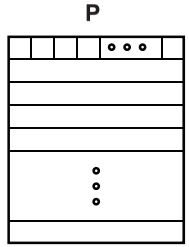
- distributed memory and directory
  - flat schemes
    - » hash
  - hierarchical schemes
    - » node's directory entry for a block says whether each subtree caches the block
    - » to find directory info, send "search" message up to parent
      - routes itself through directory lookups
    - » like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

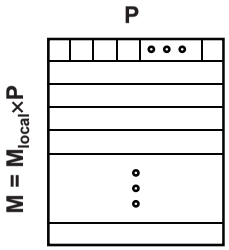
Flat, Memory-based Schemes

- 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

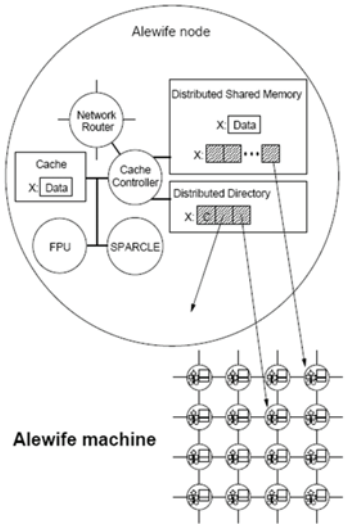
- Optimizations for full bit vector schemes
  - increase cache block size (reduces storage overhead proportionally)
  - use multiprocessor nodes (bit per mp node, not per processor)
  - still scales as P*M, but reasonable for all but very large machines
    - » 256-procs, 4 per cluster, 128B line: 6.25%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

- *Limit*ed Directory that is *L*ocally *E*xtended through *S*oftware *S*upport
- 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.

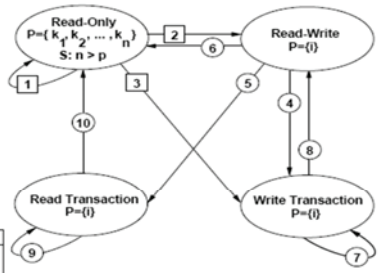
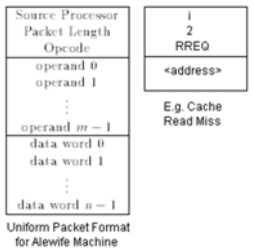
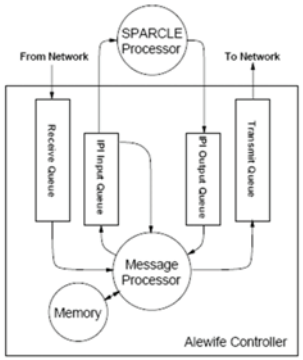


Figure 2: Directory state transition diagram.

- 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)

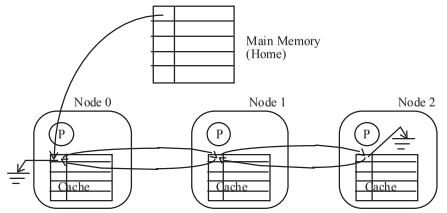
Meta State	Description
Normal	Directory handled by hardware.
Trans-In-Progress	Interlock, software processing.
Trap-On-Write	Trap: WREQ, UPDATE, REPM.
Trap-Always	Trap: all incoming packets.

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

Issues for Directory Protocols

- Correctness
- Performance
- Complexity and dealing with errors

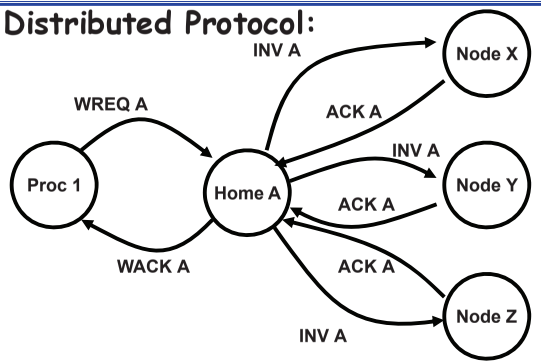
Discuss major correctness and performance issues that a protocol must address

Then delve into memory- and cache-based protocols, tradeoffs in how they might address (case studies)

Complexity will become apparent through this

Implications for Implementation

- Consider Distributed Protocol:



- Serious restrictions on when you can issue requests
- Processor held up on store until:
  - Every processor cache invalidated!
- How to prove that this is sequentially consistent??

Correctness

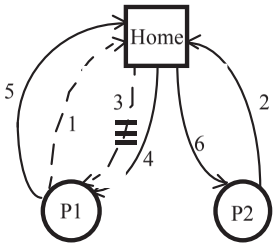
- Ensure basics of coherence at state transition level
  - relevant lines are updated/invalidated/fetched
  - correct state transitions and actions happen
- Ensure ordering and serialization constraints are met
  - for coherence (single location)
  - for consistency (multiple locations): assume sequential consistency
- Avoid deadlock, livelock, starvation
- Problems:
  - multiple copies AND multiple paths through network (distributed pathways)
  - unlike bus and non cache-coherent (each had only one)
  - large latency makes optimizations attractive
    - » increase concurrency, complicate correctness

Coherence: Serialization to a Location

- Need entity that sees op's from many procs
- bus:
  - multiple copies, but serialization imposed by bus order
  - Timestamp snooping: serialization imposed by virtual time
- scalable MP without coherence:
  - main memory module determined order
- scalable MP with cache coherence
  - home memory good candidate
    - » all relevant ops go home first
  - but multiple copies
    - » valid copy of data may not be in main memory
    - » reaching main memory in one order does not mean will reach valid copy in that order
    - » serialized in one place doesn't mean serialized wrt all copies

Serialization: Filter Through Home Node?

- Need a serializing agent
  - home memory is a good candidate, since all misses go there first
- Having single entity determine order is not enough
  - it may not know when all xactions for that operation are done everywhere



1. P1 issues read request to home node for A
2. P2 issues read-exclusive request to home corresponding to write of A. But won't process it until it is done with read
3. Home recieves 1, and in response sends reply to P1 (and sets directory presence bit). Home now thinks read is complete. Unfortunately , the reply does not get to P1 right away .
4. In response to 2, home sends invalidate to P1; it reaches P1 before transaction 3 (no point-to-point order among requests and replies).
5. P1 receives and applies invalideate, sends ack to home.
6. Home sends data reply to P2 corresponding to request 2.

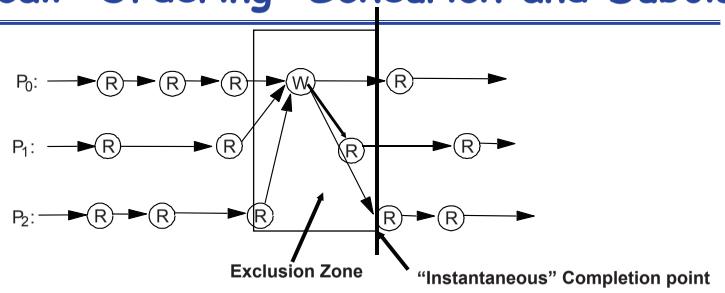
Finally , transaction 3 (read reply) reaches P1.

- Problem:
  - Home deals with write access before prev. is fully done
  - P1 should not allow new access to line until old one "done"

Basic Serialization Solution

- Use additional 'busy' or 'pending' directory *and* cache states
- Cache-side: "Transaction Buffer"
  - Similar to memory load/store buffer in uniprocessor
  - Handles misordering in network:
    - » E.g. Sent request, got invalidate first: wait to invalidate until get response from memory (either data or NACK)
    - » Make sure that memory state machine has actual understanding of state of caches + network
- Memory Side: Indicate that operation is in progress, further operations on location must be delayed or queued
  - buffer at home
  - buffer at requestor
  - NACK and retry
  - forward to dirty node

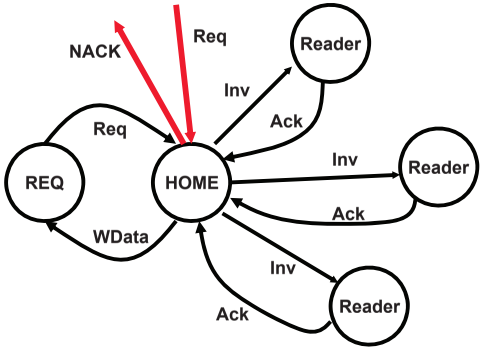
Recall: Ordering: Scheurich and Dubois



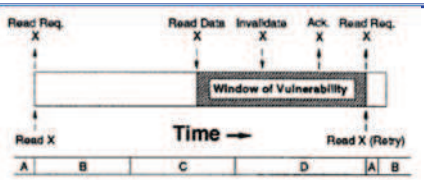
- Sufficient Conditions
  - every process issues mem operations in program order
  - after a write operation is issued, the issuing process waits for the write to complete before issuing next memory operation
  - after a read is issued, the issuing process waits for the read to complete and for the write whose value is being returned to complete (globally) before issuing its next operation
- How to get exclusion zone for directory protocol?
  - Clearly need to make sure that invalidations *really* invalidate copies
    - » Keep state to handle reordering at client (previous slide's problem)
  - While acknowledgements outstanding, cannot handle read requests
    - » NAK read requests
    - » Queue read requests

Achieving write exclusion zone

- Example for invalidation-based scheme:
  - block owner (home node) provides appearance of atomicity by waiting for all invalidations to be ack'd before allowing access to new value
  - As a result, write commit point becomes point at which WData leaves home node (after last ack received)
- Much harder in update schemes!



Livelock: Cache Side



- Window of Vulnerability: Time between return of value to cache and consumption by cache
  - Can lead to cache-side thrashing during context switching
  - Various solutions possible include stalling processor until result can be consumed by processor
  - Discussion of "Closing the Window of Vulnerability in Multiphase Memory Transactions" by Kubiatowicz, Chaiken, and Agarwal
- Solution:
  - Provide Transaction buffers to track state of outstanding requests
  - Stall context switching and invalidations selectively when thrashing detected (presumably rare)

Livelock: Memory Side

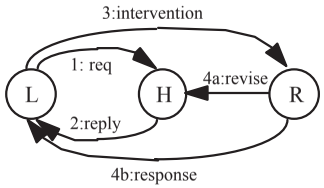
- What happens if popular item is written frequently?
  - Possible that some disadvantaged node never makes progress!
  - This is a *memory-side* thrashing problem
- Examples:
  - High-conflict lock bit
  - Scheduling queue
- Solutions?
  - Ignore
    - » Good solution for low-conflict locks
    - » Bad solution for high-conflict locks
  - Software queueing: Reduce conflict by building a queue of locks
    - » Example: MCS Lock ("Mellor-Crummey-Scott")
  - Hardware Queuing at directory: Possible scalability problems
    - » Example: QOLB protocol ("Queue on Lock Bit")
    - » Natural fit to SCI protocol
  - Escalating priorities of requests (SGI Origin)
    - » Pending queue of length 1
    - » Keep item of highest priority in that queue
    - » New requests start at priority 0
    - » When NACK happens, increase priority

Performance

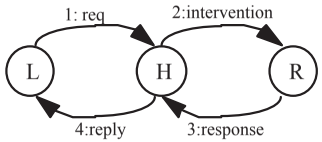
- Latency
  - protocol optimizations to reduce network xactions in critical path
  - overlap activities or make them faster
- Throughput
  - reduce number of protocol operations per invocation
  - Reduce the *residency time* in the directory controller
    - » Faster hardware, etc
- Care about how these scale with the number of nodes

Protocol Enhancements for Latency

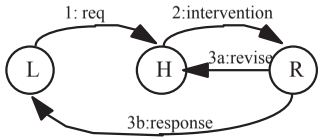
- Forwarding messages: memory-based protocols



(a) Strict request-reply



(a) Intervention forwarding



(a) Reply forwarding

Intervention is like a req, but issued in reaction to req. and sent to cache, rather than memory.

Other Latency Optimizations

- Throw hardware at critical path
  - SRAM for directory (sparse or cache)
  - bit per block in SRAM to tell if protocol should be invoked
- Overlap activities in critical path
  - multiple invalidations at a time in memory-based
  - overlap invalidations and acks in cache-based
  - lookups of directory and memory, or lookup with transaction
    - » speculative protocol operations
- Relaxing Consistency, e.g. Stanford Dash:
  - Write request when outstanding reads: home node gives data to requestor, directs invalidation acks to be returned to requester
  - Allow write to continue immediately upon receipt of data
  - Does not provide Sequential consistency (provides release consistency), but still write atomicity:
    - » Cache can refuse to satisfy intervention (delay or NACK) until all acks received
    - » Thus, writes have well defined ordering (coherence), but execution note necessarily sequentially consistent (instructions after write not delayed until write completion)

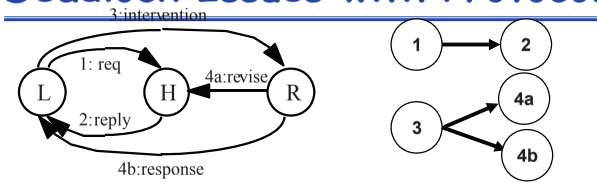
Increasing Throughput

- Reduce the number of transactions per operation
  - invals, acks, replacement hints
  - all incur bandwidth and assist occupancy
- Reduce assist occupancy or overhead of protocol processing
  - transactions small and frequent, so occupancy very important
- Pipeline the assist (protocol processing)
- Many ways to reduce latency also increase throughput
  - e.g. forwarding to dirty node, throwing hardware at critical path...

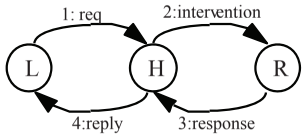
Deadlock, Livelock, Starvation

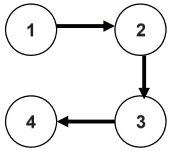
- Request-response protocol
- Similar issues to those discussed earlier
  - a node may receive too many messages
  - flow control can cause deadlock
  - separate request and reply networks with request-reply protocol
  - Or NACKs, but potential livelock and traffic problems
- New problem: protocols often are not strict request-reply
  - e.g. rd-excl generates invalidation requests (which generate ack replies)
  - other cases to reduce latency and allow concurrency

Deadlock Issues with Protocols

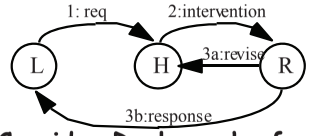


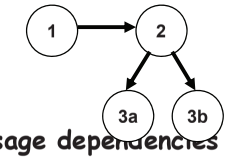
2 Networks Sufficient to Avoid Deadlock





Need 4 Networks to Avoid Deadlock

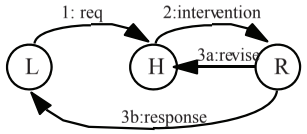


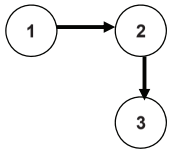


Need 3 Networks to Avoid Deadlock

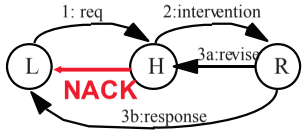
- Consider Dual graph of message dependencies
  - Nodes: Networks, Arcs: Protocol actions
  - Number of networks = length of longest dependency
  - Must always make sure response (end) can be absorbed!

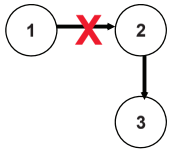
Mechanisms for reducing depth



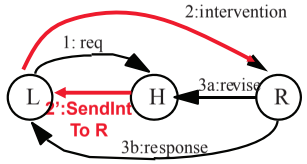


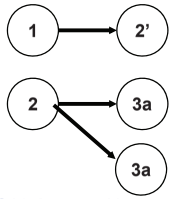
Original: Need 3 Networks to Avoid Deadlock





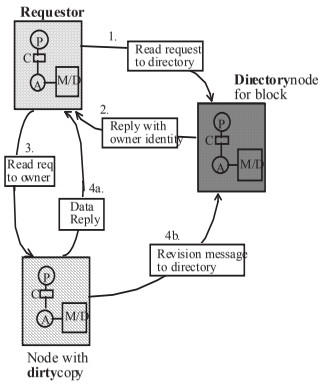
Optional NACK When blocked Need 2 Networks to



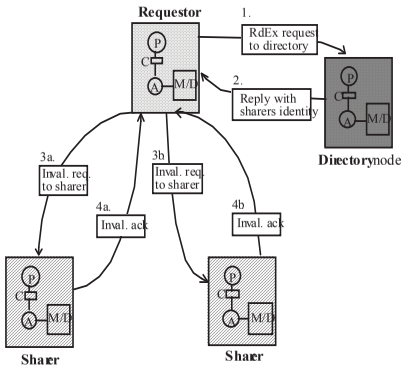


Transform to Request/Resp: Need 2 Networks to

Example of two-network protocols: Only Request-Response (2-level responses)



(a) Read miss to a block in dirty state



(b) Write miss to a block with two sharers

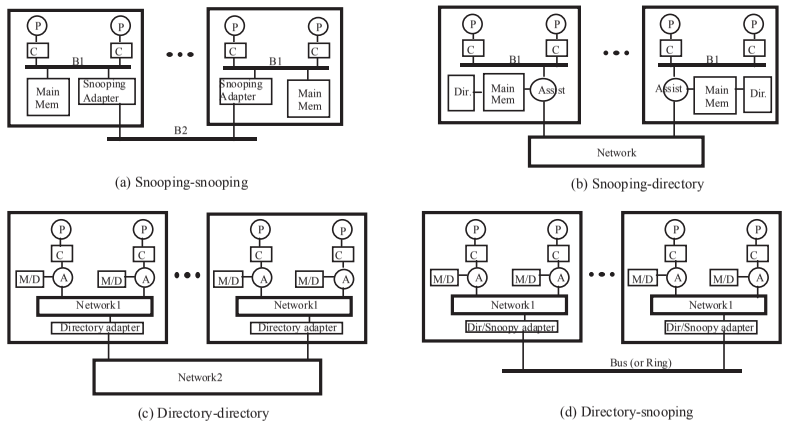
Complexity?

- Cache coherence protocols are complex
- Choice of approach
  - conceptual and protocol design versus implementation
- Tradeoffs within an approach
  - performance enhancements often add complexity, complicate correctness
    - » more concurrency, potential race conditions
    - » not strict request-reply
- Many subtle corner cases
  - BUT, increasing understanding/adoption makes job much easier
  - automatic verification is important but hard

A Popular Middle Ground

- Two-level "hierarchy"
- Individual nodes are multiprocessors, connected non-hiearchically
  - e.g. mesh of SMPs
- Coherence across nodes is directory-based
  - directory keeps track of nodes, not individual processors
- Coherence within nodes is snooping or directory
  - orthogonal, but needs a good interface of functionality
- Examples:
  - Convex Exemplar: directory-directory
  - Sequent, Data General, HAL: directory-snoopy
- SMP on a chip?

Example Two-level Hierarchies



(a) Snooping-snooping (b) Snooping-directory

(c) Directory-directory (d) Directory-snooping

Advantages of Multiprocessor Nodes

- Potential for cost and performance advantages
  - amortization of node fixed costs over multiple processors
    - » applies even if processors simply packaged together but not coherent
  - can use commodity SMPs
  - less nodes for directory to keep track of
  - much communication may be contained within node (cheaper)
  - nodes prefetch data for each other (fewer "remote" misses)
  - combining of requests (like hierarchical, only two-level)
  - can even share caches (overlapping of working sets)
  - benefits depend on sharing pattern (and mapping)
    - » good for widely read-shared: e.g. tree data in Barnes-Hut
    - » good for nearest-neighbor, if properly mapped
    - » not so good for all-to-all communication

Disadvantages of Coherent MP Nodes

- Bandwidth shared among nodes
  - all-to-all example
  - applies to coherent or not
- Bus increases latency to local memory
- With coherence, typically wait for local snoop results before sending remote requests
- Snoopy bus at remote node increases delays there too, increasing latency and reducing bandwidth
- May hurt performance if sharing patterns don't comply

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
- Snoopy Bus Protocols
  - Make use of broadcast to ensure coherence
- 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