Distributed shared memory protocols

- Distributed shared-memory machines need cache coherence for the same reasons as centralized shared-memory machines
  - Centralized protocols have drawbacks in these architectures due to the interconnection network and scalability requirements
- DSM might not use hardware mechanisms!
  - Instead, focus on scalability (Cray T3D)
  - In this scheme, only data that actually resides in the private memory may be cached (shared data is marked uncacheable)
  - Coherence is maintained by **software** => several disadvantages
  - Compiler mechanisms are very limited
    - They have to be very conservative, e.g. treat a block on another processor as dirty even though it may not be
    - This results in excessive coherence overhead (extra fetching)

Issues with DSM coherence protocols

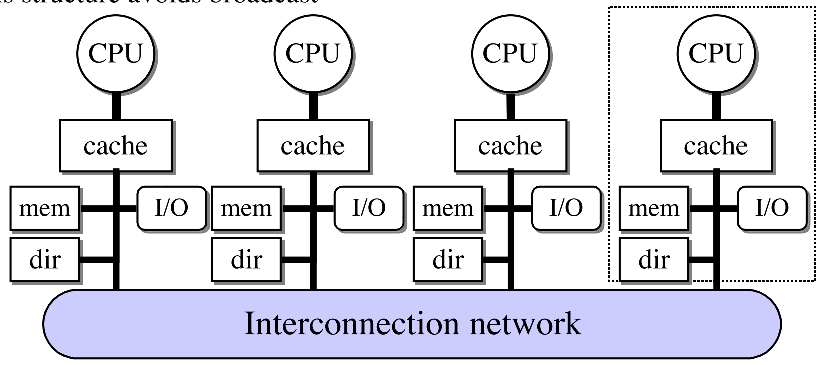
- More disadvantages of software implemented coherence
  - Multiple words in a block provide no advantage
    - Software coherence must be run each time a word is needed
    - The advantage of spatial locality (the "prefetch" of other words in the block) is lost in single word fetching
  - Latencies to remote memory are relatively high.
    - Remote references can take 50 - 1000 CPU cycles, making coherency "misses" a very costly proposition
- Snooping isn't feasible for DSM
  - Snooping coherence schemes aren't scalable => problem for DSMs
  - The distributed nature of the snooping protocol's data structure (which maintains the state of the cache blocks) does ***NOT*** scale well
  - Snooping requires broadcast (communication with all caches on every miss) => very expensive with an interconnection network

Directory-based coherence

- Alternative to snooping: directories, which hold:
  - The state of every block in memory (shared, uncached or exclusive)
    - Exclusive => the block has been written, is in one cache and memory is out-of-date
    - This information is also keep in the cache for efficiency reasons
  - Which caches have copies
    - May be implemented using a bit vector for each block with the processors identified by the bit's position
  - Whether or not the block is dirty
- The amount of information in the directory is proportional to number of processors * number of memory blocks
  - ⇒ This works O.K. for less than 100 processors -- other solutions are needed for >100 processors

Directory location

- Directory entries may be distributed along with the memory
  - High-order bits of an address can be used to find the location of a particular block of memory
  - Directory and data for a block at the same place
- This structure avoids broadcast



Directory-based cache coherence protocols

- Handle two basic primitives
  - Read miss
  - Write to a shared, clean cache block
  - Awrite miss is a combination of these two
- Simplifying assumptions still hold here
  - Writes to non-exclusive data generate write misses
  - Write misses are atomic (processors block until the access completes)
- This introduces two complications
  - There is no longer a bus => no single point of arbitration
  - Broadcast is to be avoided => the directory and cache must issue explicit response messages, e.g., *invalidate* and *write-back* request messages

Directory protocol: message types

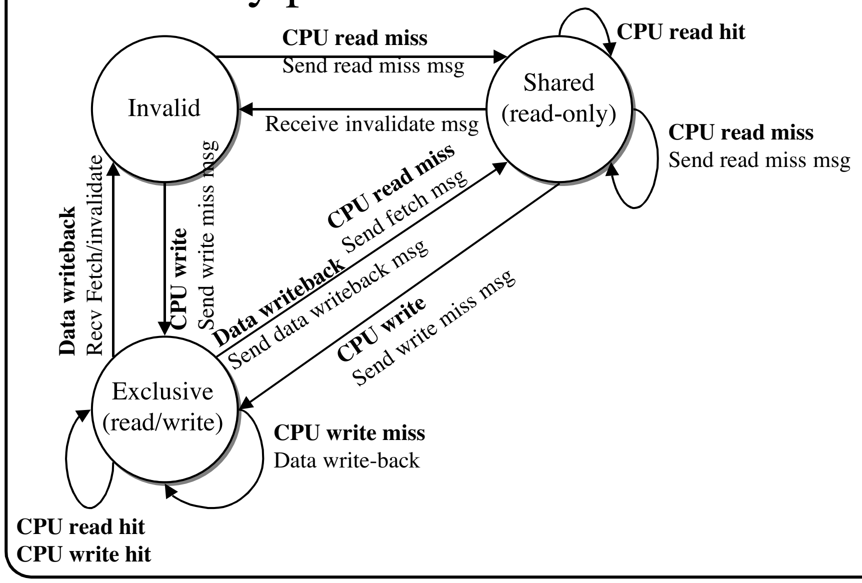
- States and transitions at each cache are identical to the snooping protocol
- Actions are somewhat different, however
- Message types for directory-based protocols
  - Local node: Where the request originates
  - Home node: Where memory and directory live
  - Remote node: Node that has a copy of the block (exclusive or shared)

Message type	Source	Destination	Contents	Function
Read miss	Local	Home	P, A	P has a read miss at addr A Request data & make P a read sharer
Write miss	Local	Home	P, A	P has a write miss at addr A Request data & make P exclusive owner
Invalidate	Home	Remote	A	Invalidate a shared copy of data at addr A

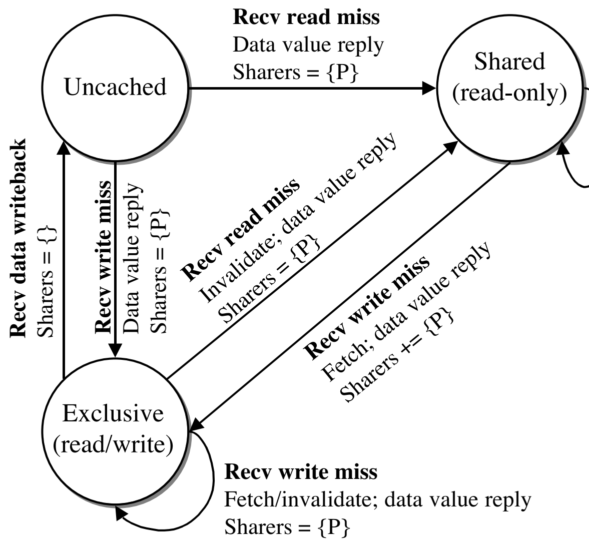
Directory protocol: more message types

Message type	Source	Destination	Contents	Function
Fetch	Home	Remote	A	Fetch block at addr A & send to home Change the state of A in remote to shared
Fetch/invalidate	Home	Remote	A	Fetch block at addr A & send to home Invalidate the block in the cache
Data value reply	Home	Local	Data	Return a data value from the home memory
Data writeback	Remote	Home	A, Data	Write back a data value for addr A

Directory protocol: cache state machine



Directory protocol: directory state machine



Directory receives only 3 types of messages
- Read miss
- Write miss
- Data writeback

Uncached & shared states

- Uncached state
  - When the block is uncached, the directory can only receive two kinds of messages: read miss and write miss
    - A read miss moves the block into the shared state
    - A write miss moves it into exclusive
  - In either case, the directory updates its list of sharing nodes to include only the node that requested the data
- Shared state
  - Again, only read or write misses are possible, since all caches have the same value as memory
  - Read miss => node requesting data is added to the list of sharing nodes
  - Write miss => block is moved to the exclusive state
    - Invalidate messages are sent to all current sharing nodes.
    - Sharing list is updated to only the requesting processor

Exclusive state

- Read miss
  - Owner is sent a fetch message telling it to write data back to memory
  - Requesting node is added to the sharing list
  - Block is marked as shared
- Write miss: block must be written back by the current owner
  - Directory sends out a fetch message
  - When the data is written, the directory forwards it to the new owner and replaces the old owner with the new owner in the sharing list
- Write-back: the data is updated in memory and the block goes into the uncached state and the sharing list is cleared
- Optimization: have the old owner send the data directly to the new owner on a write miss
  - May be done either instead of or in addition to writing the data to home

Directory protocol issues

- What happens when read-only data is replaced?
  - This scheme does not explicitly notify the directory when a clean block is replaced in the cache
  - This is OK => the cache will simply ignore invalidate messages for blocks that are not currently cached
  - Potential problem: the directory may send a few unnecessary messages
    - Probably not as bad as having the remote caches send a message each time they replace a block
- Synchronization
  - Deciding the order of accesses in a distributed memory system is harder
  - Without a shared bus, it's impossible to tell which **writes** come first
    - It's not feasible to stall all accesses until a write completes
  - Often, this can be handled by requiring all writes to be atomic
    - However, doing so slows down the system greatly

Synchronization in parallel processors

- Processors within a parallel system must have some method of synchronization
  - Software routines are usually built on top of hardware-supplied synchronization instructions
- For shared-memory machines, the key element is an uninterruptible instruction that can atomically retrieve and change a value
  - Atomic exchange: exchanges register and memory location atomically
  - Test-and-set
  - Implementation is challenging since it requires both a memory read and write to execute atomically
  - This complicates coherence and does not scale well

Load-linked & store conditional

- Another option: use a pair of instructions:

```
try: mov  R3,R4    ; move exchange value into R3
     ll   R2,0(R1) ; load the value at 0(R1) into R2
     sc   R3,0(R1) ; store the value R3 and set R3
     beqz R3,try   ; branch if value set is changed to 0
     mov  R4,R2    ; put load value into R4
```

- Store conditional (*sc*) fails if
  - Memory location specified by the *load linked* instruction is changed before the *store conditional* instruction (to the same address)
  - If it fails, the sequence is executed again
- *ll* is implemented by storing the address given in the instruction in a link register
  - If an interrupt occurs or if the cache block containing the address is invalidated, the *ll* register is set to 0 and *sc* fails

Implementing locks using coherence

- How can locks be cached in machines with cache coherence?
  - Spin-lock operation is performed on a local cached copy
    - This reduces memory bandwidth
  - There is often locality in lock access => caching reduces time to acquire the lock
- Assume that we have an exchange instruction
  - To implement a spin-lock, use the following where 0 indicates success:

```
           lwi  R2,#1     ; load immediate #1
lockit: exch R2,0(R1)  ; exchange R2 with 0(R1)
        bnez R2,lockit ; if 1 returned, fail
```

- Problem:
  - Each *exch* operation requires a write
  - Most writes result in a write miss => writing requires exclusive access

Implementing locks using coherence

- In the loop, read instead and write only when the lock is free

```
lockit: lw   R2,0(R1)  ; read the lock
        bnez R2,lockit ; keep reading if lock not free
        lwi  R2,#1     ; load the lock value
        exch R2,0(R1)  ; race to exchange & get 0
        bnez R2,lockit ; if another processor beat us, start over
```

- A load linked/store conditional version need not cause any bus traffic during the testing operation:

```
lockit: ll   R2,0(R1)  ; read the lock
        bnez R2,lockit ; keep reading if lock not free
        lwi  R2,#1     ; load the lock value
        sc   R2,0(R1)  ; Try to store & get 0
        beqz R2,lockit ; if another processor beat us, start over
```

  - However, when the lock is released, a lot of traffic is generated
  - This makes it difficult to scale this implementation to many processors

Barrier synchronization

- Barrier synchronization: another common synchronization operation in programs with parallel loops
  - Allows multiple processes on multiple CPUs to wait until a certain number of processes have reached a “barrier”
  - When sufficient processes arrive, all waiting processes may continue

```
lock(counterlock);         // ensure count update is atomic
if (count == 0)            // First process to barrier -- reset 'release'
  release = 0;
count += 1;                // Count arrivals
unlock(counterlock);       // Release lock
if (count == total) {      // All have arrived, so reset
  count = 0;               // counter and release processes
  release = 1;
} else {                   // Waiting for more processes
  spin (release == 1);     // Spin until release set to 1
}
```

Barrier synchronization

- Previous method can fail if one process ‘races’ ahead and resets release before the last process has been scheduled again and exits the spin
- Instead, use a sense-reversing barrier
  - A fix which uses private per-process variables.
  - local_sense is initialized to 1 for each process

```
Local_sense = !local_sense;   // toggle private variable
lock(counterlock);            // ensure count update is atomic
if (count == 0)               // First process to barrier -- reset 'release'
  release = 0;
count += 1;                   // Count arrivals
unlock(counterlock);          // Release lock
if (count == total) {         // All have arrived, so reset
  count = 0;                  // counter and release processes
  release = local_sense;
} else {                      // Waiting for more processes
  spin (release == local_sense);
}
```

Memory consistency models

- Deciding when a change to memory should be propagated to other CPUs is a difficult problem
  - Message-passing machines require explicit propagation, so the decision is left entirely to software
  - Shared memory in hardware has to support hardware mechanisms for making this decision

Future of parallel processors

- This class has only scratched the surface on multiprocessors
  - There is more than enough material to spend an entire semester discussing MP hardware design
  - Predicting the future of MPs is even harder than understanding how today’s MPs work
- Large scale machines that scale up naturally
  - These would be built from commodity elements that can be added in small numbers to build a a big system
  - The interconnect is proprietary, but the processor is commodity
  - The SGI Origin 2000 series and Cray T3E work this way

Future of parallel processors

- Large-scale machines built of clusters of mid-range machines
  - Require faster uniprocessors but can be built from fewer machines
  - Examples include the SGI Challenge and Sun Enterprise
- Off-the-shelf nodes with custom interconnect
  - Uses standard processor boards
  - Uses a high-speed custom-built interconnect
  - Examples include IBM SP/2
- All off-the-shelf components
  - Everything (processor, network) is standard and relatively inexpensive
  - Workstation clusters (Beowulf) are a good example of this