Outline

- Directory-Based Cache Coherence
  - Review
  - Basic Idea
  - Some Variations

- SGI Origin 2000 Case Study

- Memory Consistency Models Revisited

- Advanced Topics
Review: Snooping Coherence

- Controller updates state of blocks in response to processor and snoop events and generates bus transactions
- Often have duplicate cache tags
- Snoopy protocol
  - set of states
  - state-transition diagram
  - actions
- Basic Choices
  - write-through vs. write-back
  - invalidate vs. update

Review: MSI State Diagram
Large Scale Shared Memory Multiprocessors

- 100s to 1000s of nodes (processors) with single shared physical address space
- Use General Purpose Interconnection Network
  - Still have cache coherence protocol
  - Use messages instead of bus transactions
  - No hardware broadcast
- Communication Assist

Directory Based Cache Coherence

- Avoid broadcast request to all nodes on a miss
  - traffic
  - time
- Maintain directory of which nodes have cached copies of the block (directory controller + directory state)
- On a miss, send message to directory
  - communication assist
- Directory determines what (if any) protocol action is required
  - e.g., invalidation
- Directory waits for protocol actions to finish and then responds to the original request
Directory Example

Centralized Directory

- **Single directory** that contains a copy of all nodes' cache tags

**Problems**
- Bottleneck (1000s of processors...)
- Directory changes with number of nodes

**Positives**
- Send Invalidates/Updates only to nodes that have a copy of block
Distributed Directory

- Distribute Directory among memory modules
- Maintain directory for each memory block
  - memory block = coherence block size: block's home node = node with directory

Directory Nomenclature

- Dir\(_X\)
- Directory of \(i\) pointers (\(i \leq \) Total number of nodes)
- \(X\) specifies what to do on Shared to Modified transition
  - \(B\) => Broadcast
  - \(NB\) => No Broadcast
  - \(SW\) => Software
- \(\text{Dir}_N\) = full-map directory
  - Bit vector per memory block
  - Bit per node in system
  - No need to broadcast
Coarse Vector and Sparse Directories

Coarse Vector
• Instead of full-map or broadcast, indicate a set of nodes that may have the block
• Reduces space requirements
• Many applications have near neighbor sharing

Sparse
• Not all of memory is in processor caches
• Cache of directory entries at memory

Outline

• Directory-Based Cache Coherence

• SGI Origin 2000 Case Study
  – Overview
  – Directory & Protocol States
  – Detailed Coherence Protocol Examples

• Memory Consistency Models Revisited

• Advanced Topics
Origin2000 System Overview

- Upto 512 nodes (1024 processors)
- Directory state in same or separate DRAMs, accessed in parallel

Origin Node Board

- Hub is 500K-gate in 0.5 u CMOS
- Has outstanding transaction buffers for each processor (4 each)
- Has two block transfer engines (memory copy and fill)
- Interfaces to and connects processor, memory, network and I/O
- Provides support for synch primitives, and for page migration
- Two processors within node not snoopy-coherent (cost)
Origin Network

- Each router has six pairs of unidirectional links
  - Two to nodes, four to other routers
  - Latency: 41ns pin to pin across a router
- Flexible cables up to 3 ft long
- Four “virtual channels”: request, reply, two for priority or I/O

Origin Directory Structure

- Flat, Memory based: all directory information at the home
- Three directory formats:
  - (1) if exclusive in a cache, entry is pointer to that specific processor (not node)
  - (2) if shared, bit vector: each bit points to a node (Hub), not processor
  - Invalidation sent to a Hub is broadcast to both processors in the node
  - Two sizes, depending on scale
    - 16-bit format (32 procs), kept in main memory DRAM
    - 64-bit format (128 procs), extra bits kept in extension memory
  - (3) for larger machines, coarse vector: each bit corresponds to p/64 nodes
- Ignore coarse vector in discussion for simplicity
Origin Cache and Directory States

- **Cache states:** MESI
- **Seven directory states**
  - *unowned:* no cache has a copy, memory copy is valid
  - *shared:* one or more caches has a shared copy, memory is valid
  - *exclusive:* one cache (pointed to) has block in modified or exclusive state
  - three *pending* or *busy* states, one for each of the above:
    » indicates directory has received a previous request for the block
    » couldn’t satisfy it itself, sent it to another node and is waiting
    » cannot take another request for the block yet
  - *poisoned* state, used for efficient page migration (later)
- **Let’s see how it handles read and “write” requests**
  - no point-to-point order assumed in network

Handling a Read Miss

- **Hub looks at address**
  - if remote, sends request to home
  - if local, looks up directory entry and memory itself
  - directory may indicate one of many states
- **Shared or Unowned State:**
  - if shared, directory sets presence bit
  - if unowned, goes to exclusive state and uses pointer format
  - replies with block to requestor
    » strict request-reply (no network transactions if home is local)
  - actually, also looks up memory speculatively to get data
    » directory lookup returns one cycle earlier
  - if directory is shared or unowned, data already obtained by Hub
  - if not one of these, speculative memory access is wasted
- **Busy state: not ready to handle**
  - NACK, so as not to hold up buffer space for long
Read Miss to Block in Exclusive State

- Most interesting case
  - if owner is not home, need to get data to home and requestor from owner
  - Uses reply forwarding for lowest latency and traffic
    - not strict request-reply

\[
\begin{array}{c}
L \quad 1: \text{req} \quad 2b: \text{intervention} \\
H \quad 2a: \text{spec. reply} \quad 3b: \text{response} \\
R
\end{array}
\]

Actions at Home and Owner

- At the home:
  - set directory to busy state and NACK subsequent requests
    - general philosophy of protocol
    - can’t set to shared or exclusive
    - alternative is to buffer at home until done, but input buffer problem
  - set and unset appropriate presence bits
  - assume block is clean-exclusive and send speculative reply

- At the owner:
  - If block is dirty
    - send data reply to requestor, and “sharing writeback” with data to home
  - If block is clean exclusive
    - similar, but don’t send data (message to home is called “downgrade”)

- Home changes state to shared when it receives msg
Handling a Write Miss

- Request to home could be upgrade or read-exclusive
- State is busy: NACK
- State is unowned:
  - if RdEx, set bit, change state to dirty, reply with data
  - if Upgrade, means block has been replaced from cache and directory already notified, so upgrade is inappropriate request
    » NACKed (will be retried as RdEx)
- State is shared or exclusive:
  - invalidations must be sent
  - use reply forwarding; i.e. invalidations acks sent to requestor, not home

Write to Block in Shared State

- At the home:
  - set directory state to exclusive and set presence bit for requestor
    » ensures that subsequent requests will be forwarded to requestor
  - If RdEx, send “excl. reply with invals pending” to requestor (contains data)
    » how many sharers to expect invalidations from
  - If Upgrade, similar “upgrade ack with invals pending” reply, no data
  - Send invals to sharers, which will ack requestor
- At requestor, wait for all acks to come back before “closing” the operation
  - subsequent request for block to home is forwarded as intervention to requestor
  - for proper serialization, requestor does not handle it until all acks received for its outstanding request
Write to Block in Exclusive State

- If upgrade, not valid so NACKed
  - another write has beaten this one to the home, so requestor's data not valid
- If RdEx:
  - like read, set to busy state, set presence bit, send speculative reply
  - send invalidation to owner with identity of requestor
- At owner:
  - if block is dirty in cache
    » send "ownership xfer" revision msg to home (no data)
    » send response with data to requestor (overrides speculative reply)
  - if block in clean exclusive state
    » send "ownership xfer" revision msg to home (no data)
    » send ack to requestor (no data; got that from speculative reply)

Handling Writeback Requests

- Directory state cannot be shared or unowned
  - requestor (owner) has block dirty
  - if another request had come in to set state to shared, would have been forwarded to owner and state would be busy
- State is exclusive
  - directory state set to unowned, and ack returned
- State is busy: interesting race condition
  - busy because intervention due to request from another node (Y) has been forwarded to the node X that is doing the writeback
    » intervention and writeback have crossed each other
  - Y's operation is already in flight and has had it's effect on directory
  - can't drop writeback (only valid copy)
  - can't NACK writeback and retry after Y's ref completes
    » Y's cache will have valid copy while a different dirty copy is written back
Solution to Writeback Race

- Combine the two operations
- When writeback reaches directory, it changes the state
  - to shared if it was busy-shared (i.e. Y requested a read copy)
  - to exclusive if it was busy-exclusive
- Home forwards the writeback data to the requestor Y
  - sends writeback ack to X
- When X receives the intervention, it ignores it
  - knows to do this since it has an outstanding writeback for the line
- Y’s operation completes when it gets the reply
- X’s writeback completes when it gets the writeback ack

Replacement of Shared Block

- Could send a replacement hint to the directory
  - to remove the node from the sharing list
- Can eliminate an invalidation the next time block is written
- But does not reduce traffic
  - have to send replacement hint
  - incurs the traffic at a different time
- Origin protocol does not use replacement hints
- Total transaction types:
  - coherent memory: 9 request transaction types, 6 inval/intervention, 39 reply
  - noncoherent (I/O, synch, special ops): 19 request, 14 reply (no inval/intervention)
Preserving Sequential Consistency

- R10000 is dynamically scheduled
  - memory operations speculatively execute out of program order
  - but ensures that they become visible and complete in order
  - doesn’t satisfy sufficient conditions, but provides SC

- An interesting issue w.r.t. preserving SC
  - On a write to a shared block, requestor gets two types of replies:
    » exclusive reply from the home, indicates write is serialized at memory
    » invalidation acks, indicate that write has completed wrt processors
  - But microprocessor expects only one reply (as in a uniprocessor)
    » so replies have to be dealt with by requestor’s HUB
  - To ensure SC, Hub must wait till inval acks are received before replying to proc
    » can’t reply as soon as exclusive reply is received
      • would allow later accesses from proc to complete (writes become visible) before this write

Outline

- Directory-Based Cache Coherence
- SGI Origin 2000 Case Study
- Memory Consistency Models Revisited
  - Review SC
  - Relax ordering from writes to reads
  - Relaxing all order
- Advanced Topics
Review: Coherence vs. Consistency

- Intuition says loads should return latest value
  - what is latest?
- Coherence concerns only one memory location
- Consistency concerns apparent ordering for all locations
- A Memory System is Coherent if
  - can serialize all operations to that location such that,
  - operations performed by any processor appear in program order
    - program order = order defined program text or assembly code
  - value returned by a read is value written by last store to that location

Review: Why Coherence != Consistency

/* initial A = B = flag = 0 */
P1
A = 1; while (flag == 0); /* spin */
B = 1; print A;
flag = 1; print B;

Intuition says printed A = B = 1
Coherence doesn’t say anything, why?
Review: Sequential Consistency (SC)

Sequential processors issue memory ops in program order.

P1 P2 P3

Memory

Switch randomly set after each memory op provides single sequential order among all operations.

Review: Sufficient Conditions for SC

- Every processor issues memory ops in program order.
- Processor must wait for store to complete before issuing next memory operation.
- After load, issuing proc waits for load to complete, and store that produced value to complete before issuing next op.
- Easily implemented with shared bus.
Relaxed Memory Models

- **Motivation with Directory Protocols**
  - Misses have longer latency (do cache hits to get to next miss)
  - Collecting acknowledgements can take even longer

- **Recall SC has**
  - Each processor generates at total order of its reads and writes
    \( (R\rightarrow R, R\rightarrow W, W\rightarrow W, & W\rightarrow R) \)
  - That are interleaved into a global total order

- **(Most) Relaxed Models**
  - **PC:** Relax ordering from writes to (other proc’s) reads
  - **RC:** Relax all read/write orderings (but add “fences”)

Relax Write to Read Order

```c
/* initial A = B = 0 */

P1
A = 1;
r1 = B;

P2
B = 1;
r2 = A;
```

**Processor Consistent (PC) Models**
- Allow \( r1==r2==0 \) (precluded by SC)
- Examples: IBM 370, Sun TSO, & Intel IA-32
- Why do this?
  - Allows FIFO write buffers
  - Does not astonish programmers (too much)
Write Buffers w/ Read Bypass

Also Want “Causality”

/* initially all 0 */

P1
A = 1; while (flag1==0) {}; flag1 = 1;

P2
while (flag1==0) {}; flag2 = 1;

P3
r3 = A;

All commercial models guarantee causality
Why Not Relax All Order?

/* initially all 0 */

```
 P1            P2
 A = 1;        while (flag == 0); /* spin */
 B = 1;        r1 = A;
 flag = 1;     r2 = B;
```

Reorder of “A = 1”/“B = 1” or “r1 = A”/“r2 = B”
Via OOO processor, non-FIFO write buffers, delayed
directory acknowledgements, etc.

But Must Order

“A = 1”/“B = 1” before “flag = 1”
“flag != 0” before “r1 = A”/“r2 = B”

Order with “Synch” Operations

/* initially all 0 */

```
 P1            P2
 A = 1;        while (SYNCH flag == 0);
 B = 1;        r1 = A;
 SYNCH flag = 1; r2 = B;
```

Called “weak ordering” of “weak consistency” (WC)
Alternatively, relaxed consistency (RC) specializes
Acquires: force subsequent reads/writes after
Releases: force previous reads/writes before
Weak Ordering Example

Release Consistency Example
Commercial Models use “Fences”

/* initially all 0 */

P1
A = 1;
B = 1;
FENCE;
flag = 1;

P2
while (flag == 0);
FENCE;
r1 = A;
r2 = B;

Examples: Compaq Alpha, IBM PowerPC, & Sun RMO

Can specialize fences (e.g., RMO)

The Programming Interface

• WO and RC require synchronized programs
• All synchronization operations must be labeled and visible to the hardware
  – easy if synchronization library used
  – must provide language support for arbitrary ld/st synchronization (event notification, e.g., flag)
• Program that is correct for TSO portable to WO & RC
Outline

• Directory-Based Cache Coherence
• SGI Origin 2000 Case Study
• Memory Consistency Models Revisited
• Advanced Topics
  – Cache-Only Memory Architecture (COMA)
  – Paged-Based Distributed Shared Memory
  – Simple-COMA (S-COMA)
  – Hierarchical Coherence
  – Latency Tolerance

Review

• Recall Part 1
  – Directory-Based Cache Coherence
  – SGI Origin 2000 Case Study
  – Memory Consistency Models Revisited

• Basic Idea
  – Per-processor cache hierarchies
  – Directory interleaved with memory

• But
  – Limited capacity for replication
  – High design & implementation cost
  – Single hard-wired protocol
  – Limitations of shared physical address space
Cache Only Memory Architecture (COMA)

- Make all memory available for migration & replication
- All memory is DRAM cache called Attraction Memory

Examples
- Data Diffusion Machine (next)
- Flat COMA (fixed home for directory but not data)
- KSR-1 (hierarchy of snooping rings)

But how do you
- Find data?
- Deal with replacements?

COMA Example: Data Diffusion Machine (DDM)

- All hardware COMA
- Attraction Memory => One giant hardware cache
- Maintains both address tags and state
- Data addressed, allocated, & kept coherent in blocks
- Directory info on a per cache-block basis
- Not Home Based:
  - data is migratory => AM attracts data
  - must find a home when replacing the data
  - must find the directory entry before finding the data
DDM Directory

- Directory is hierarchical in a tree form
- Each is a set-associative cache if directory info
- Tree maintains inclusion:
  - Higher levels keep replica of lower sub-trees

```
  D
 /  \
D  D
 /  \
D  D
 /  \
D  D
```

DDM Coherence/Placement Protocol

- Simple write-invalidate protocol
- Cache states: Invalid, Exclusive, Shared
- Must traverse the directory:
  - to find a copy on a read or write miss
  - to invalidate on a write to Shared
- Directory is hierarchical set-associative caches
  - Q1: Is the block in my sub-tree?
  - Q2: Does the block exist outside my sub-tree?
  - Request goes up until Q2=no and then down
  - Request goes down until Q1=no or leaf
- On a replacement:
  - for an Exclusive copy, must find another home (HARD!)
  - for a Shared copy, must make sure other copies exist
  - else must find another home
Page Based DSM (Shared Virtual Memory)

- Forget all this hardware!

- Implemented shared virtual address space
  - On separate computers networked together
  - Use virtual memory system to do coherence on pages
  - No special hardware; no shared physical address space

- Called
  - Shared Virtual Memory (SVM) in original paper [Li & Hudak]
  - Now called Page-Based (or Software) Distributed Shared Memory

Example

- P1 read virtual address x
- Page fault
- Allocate physical frame for page(x)
- Request page(x) from home(x)
- Set readable page(x)
- Resume program

Problems
  - False-sharing
  - Page fault overhead

Advantages
  - Software Coherence protocol
  - Low cost
Relaxed Consistency

- Release consistency can delay invalidates, but HW sends them out as they occur
- Send invalidations/updates (write notices) only at synchronization
- First reference after synchronization misses

Multiple Writer Protocol

- x & y on same page P1 writes x, P2 writes y
- Don’t want delays associated with constraint of exclusive access
- Allow each processor to modify its local copy of a page between synchronization points
- Make things consistent at synchronization point
Simple COMA (S-COMA)

- **COMA**
  - Block granularity to find/allocate/replace (complex hardware)
  - Block granularity for coherence/transfers (good for false sharing)

- **Software DSM**
  - Page granularity to find/allocate/replace (use VM: good)
  - Page granularity for coherence/transfers (bad for false sharing)

- **Simple COMA**
  - Page granularity to find/allocate/replace (use VM: good)
  - Block granularity for coherence/transfers (good for false sharing)
  - Blocks act like sub-blocks on page

Sun Wildfire

- [Hagersten/Koster HPCA99]
- Begin with up to four SMP nodes
- Add pseudo-processor board to each as proxy for rest of system
- Can run CC-NUMA directory protocol
- Can selectively use S-COMA (called Coherent Memory Replication)
- Selects between with competitive algorithm [Falsafi/Wood ISCA97]
- A hierarchical methods of building parallel machines
Hierarchical Coherence

- Most solutions so far are flat
  - E.g., a directory that points to 1K processors

- Use hierarchy
  - Intra-node coherence (e.g., snooping in SMP node)
  - Inter-node coherence (e.g., directory between nodes)

- Why?
  - Divide & conquer markets (e.g., sell node)
  - Divide & conquer complexity (but must interface protocols)

Example Two-level Hierarchies

(a) Snooping-snooping
(b) Snooping-directory
(c) Directory-directory
(d) Directory-snooping
Advantages of Multiprocessor Nodes

- amortization of node fixed costs over multiple processors
- can use commodity SMPs
- less nodes for directory to keep track of
- much communication may be contained within node (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)

Disadvantages of Coherent MP Nodes

- Bandwidth shared among nodes
- Bus increases latency to local memory
- With coherence, typically wait for local snoop results before sending remote requests
- Snoopy bus at remote node increases delays there too, increasing latency and reducing bandwidth
- Overall, may hurt performance if sharing patterns don’t comply
Latency Tolerance

- **Motivation**
  - Microprocessors getting faster faster than DRAM
  - DRAM access is now 100s of “instruction opportunities”
  - Worse for parallel machines

- **Of course**
  - Use memory hierarchy
  - Make each level as fast as possible
  - make each level better (e.g., bigger)

- **Eventually**
  - Must tolerate latency for additional parallelism
  - Old idea: multitasking for disk I/O

Latency Tolerance for Message Passing

- **Block Data Transfer**
  - Of course -- messages

- **Pre-communication**
  - If possible
  - And must buffer

- **Proceeding Past Long-Latency Events**
  - Of course -- messages

- **Multithreading within Node**
  - Possible
  - But makes programming model more complex
    (intra-node communication message-passing or shared-memory?)
Latency Tolerance for Shared Memory 1 of 2

• Block Data Transfer
  – Harder to integrate
  – DMA engine (how virtualize?)
  – block load/store instructions
  – coalescing write buffer

• Pre-communication
  – Mostly prefetching
  – binding or non-binding (latter with coherence only)
  – initiated by hardware or software
    » what to prefetch
    » went to prefetch
  – TLB misses, page faults, & invalid exceptions
  – KSR-1 post-store -- a sender-initiated prefetch
  – Snarfing, deliver, update -- store allocation at destination hard

Latency Tolerance for Shared Memory 2 of 2

• Proceeding Past Long-Latency Events
  – Overlapping memory operations
  – Overlapping coherence operations
  – relaxed memory consistency models
    (or speculative implementation of strong model)

• Multithreading
  – Between instructions: HEP & Tera
    » too many contexts & fails to exploit caches
  – On cache miss: MIT Alewife & IBM RS64 (RS/6000)
    » Few contexts for in-order core
  – Within cycle: Washington Simultaneous Multithreading (SMT)
    » Good match to out-of-order core with register renaming
      (since backend need not know multiple threads exist)
Outline

- Directory-Based Cache Coherence
- SGI Origin 2000 Case Study
- Memory Consistency Models Revisited
- Advanced Topics
  - Cache-Only Memory Architecture (COMA)
  - Paged-Based Distributed Shared Memory
  - Simple-COMA (S-COMA)
  - Hierarchical Coherence
  - Latency Tolerance