What is (Hardware) Shared Memory?

- Take multiple microprocessors
- Implement a memory system with a single global physical address space (usually)
  - Communication assist HW does the “magic” of cache coherence
- Goal 1: Minimize memory latency
  - Use co-location & caches
- Goal 2: Maximize memory bandwidth
  - Use parallelism & caches

Some Memory System Options

- Bus-based shared memory
- Interconnection network

Outline

- Motivation for Cache-Coherent Shared Memory
- Snooping Cache Coherence (Chapter 5)
  - Basic systems
  - Design tradeoffs
- Implementing Snooping Systems (Chapter 6)
- Advanced Snooping Systems
Cache Coherence

- According to Webster’s dictionary …
  - Cache: a secure place of storage
  - Coherent: logically consistent

- Cache Coherence: keep storage logically consistent
  - Coherence requires enforcement of 2 properties
    1) Write propagation
       - All writes eventually become visible to other processors
    2) Write serialization
       - All processors see writes to same block in same order

Why Cache Coherent Shared Memory?

- Pluses
  - For applications - looks like multitasking uniprocessor
  - For OS - only evolutionary extensions required
  - Easy to do communication without OS
  - Software can worry about correctness first and then performance

- Minuses
  - Proper synchronization is complex
  - Communication is implicit so may be harder to optimize
  - More work for hardware designers (i.e., us!)

- Result
  - Symmetric Multiprocessors (SMPs) are the most successful parallel machines ever
  - And the first with multi-billion-dollar markets!

In More Detail

- Efficient naming
  - Virtual to physical mapping with TLBs
  - Ability to name relevant portions of objects

- Easy and efficient caching
  - Caching is natural and well-understood
  - Can be done in HW automatically

- Low communication overhead
  - Low overhead since protection is built into memory system
  - Easy for HW (i.e., cache controller) to packetize requests and replies

- Integration of latency tolerance
  - Demand-driven: consistency models, prefetching, multithreading
  - Can extend to push data to PEs and use bulk transfer

Symmetric Multiprocessors (SMPs)

- Multiple microprocessors

- Each has a cache (or multiple caches in a hierarchy)

- Connect with logical bus (totally-ordered broadcast)
  - Physical bus = set of shared wires
  - Logical bus = functional equivalent of physical bus

- Implement Snooping Cache Coherence Protocol
  - Broadcast all cache misses on bus
  - All caches “snoop” bus and may act (e.g., respond with data)
  - Memory responds otherwise
Outline

• Motivation for Cache-Coherent Shared Memory
• Snooping Cache Coherence (Chapter 5)
  – Basic systems
  – Design tradeoffs
• Implementing Snooping Systems (Chapter 6)
• Advanced Snooping Systems

Cache Coherence Problem (Step 1)

Cache Coherence Problem (Step 2)

Cache Coherence Problem (Step 3)
Snooping Cache-Coherence Protocols

- Bus provides serialization point (more on this later)
- Each cache controller "snoops" all bus transactions
  - Transaction is relevant if it is for a block this cache contains
  - Take action to ensure coherence
    - Invalidate
    - Update
    - Supply value to requestor if Owner
  - Actions depend on the state of the block and the protocol
- Main memory controller also snoops on bus
  - If no cache is owner, then memory is owner
- Simultaneous operation of independent controllers

Snooping Design Choices

- Controller updates state of blocks in response to processor and snoop events and generates bus transactions
- Often have duplicate cache tags
- Snooping protocol
  - Set of states, set of events
  - State-transition diagram
  - Actions
- Basic Choices
  - Write-through vs. write-back
  - Invalidate vs. update
  - Which states to allow

Simple 2-State Invalidate Snooping Protocol

- Write-through, no-write-allocate cache
- Proc actions: Load, Store
- Bus actions: GETS, GETX

A 3-State Write-Back Invalidation Protocol

- 2-State Protocol
  - Simple hardware and protocol
  - Uses lots of bandwidth (every write goes on bus!)
- 3-State Protocol (MSI)
  - Modified
    - One cache exclusively has valid (modified) copy Owner
    - Memory is stale
  - Shared
    - >= 1 cache and memory have valid copy (memory = owner)
    - Invalid (only memory has valid copy and memory is owner)
- Must invalidate all other copies before entering modified state
- Requires bus transaction (order and invalidate)
**MSI Processor and Bus Actions**

- **Processor:**
  - Load
  - Store
  - Writeback on replacement of modified block

- **Bus**
  - GetShared (GETS): Get without intent to modify, data could come from memory or another cache
  - GetExclusive (GETX): Get with intent to modify, must invalidate all other caches' copies
  - PutExclusive (PUTX): cache controller puts contents on bus and memory is updated
  - Definition: cache-to-cache transfer occurs when another cache satisfies GETS or GETX request

- Let's draw it!

**An MSI Protocol Example**

<table>
<thead>
<tr>
<th>Proc Action</th>
<th>P1 State</th>
<th>P2 state</th>
<th>P3 state</th>
<th>Bus Act</th>
<th>Data from</th>
</tr>
</thead>
<tbody>
<tr>
<td>initially</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. P1 load u</td>
<td>I→S</td>
<td>I</td>
<td>I</td>
<td>GETS</td>
<td>Memory</td>
</tr>
<tr>
<td>2. P3 load u</td>
<td>S</td>
<td>I</td>
<td>I</td>
<td>GETS</td>
<td>Memory</td>
</tr>
<tr>
<td>3. P3 store u</td>
<td>S→M</td>
<td>I</td>
<td>S→M GETS</td>
<td>Memory or P1 (?)</td>
<td></td>
</tr>
<tr>
<td>4. P1 load u</td>
<td>I→S</td>
<td>I</td>
<td>M→S GETS</td>
<td>P3's cache</td>
<td></td>
</tr>
<tr>
<td>5. P2 load u</td>
<td>S</td>
<td>I→S GETS</td>
<td>S</td>
<td>GETS</td>
<td>Memory</td>
</tr>
</tbody>
</table>

- Single writer, multiple reader protocol
- Why Modified to Shared in line 4?
- What if not in any cache? Memory responds
- Read then Write produces 2 bus transactions
  - Slow and wasteful of bandwidth for a common sequence of actions

**4-State (MESI) Invalidation Protocol**

- Often called the **Illinois protocol**
  - Modified (dirty)
  - Exclusive (clean unshared) only copy, not dirty
  - Shared
  - Invalid

- Requires **shared** signal to detect if other caches have a copy of block
  - I→S if shared signal is asserted by any other cache
  - I→E otherwise

- **E→M** transition doesn’t require bus transaction
  - Why is this a good thing?
More Generally: MOESI

- [Sweazey & Smith, ISCA86]
- M - Modified (dirty)
- O - Owned (dirty but shared) **WHY?**
- E - Exclusive (clean unshared) only copy, not dirty
- S - Shared
- I - Invalid

**Variants**
- MSI
- MESI
- MOESI
- MOESI

4-State Write-back Update Protocol

- Dragon (Xerox PARC)

**States**
- Exclusive (E): one copy, clean, memory is up-to-date
- Shared-Clean (SC): could be two or more copies, memory unknown
- Shared-Modified (SM): could be two or more copies, memory stale
- Modified (M)

- On a write, send new data to update other caches
- Adds Update Transaction
  - Do not confuse Update with Upgrade!
- Adds Cache Controller Update operation
- Must obtain bus before updating local copy
  - Bus is serialization point important for consistency (later ...)

Tradeoffs in Protocol Design (C&S 5.4)

- **Design issues:**
  - Which state transitions
  - What types of bus transactions
  - Cache block size
  - Cache associativity
  - Write-back vs write-through caching
  - Etc.

- **Methodology: count protocol state transitions**
  - Can then compute bandwidth, miss rates, etc.
  - Results depend on workload (diff workload ➔ diff transition rates)

Computing Bandwidth

- **Why bandwidth?**
- **Monitor state transitions**
  - Count bus transactions
  - I know how many bytes each bus transaction requires
**MESI State Transitions and Bandwidth**

<table>
<thead>
<tr>
<th>FROM/TO</th>
<th>NP</th>
<th>I</th>
<th>E</th>
<th>S</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP</td>
<td>--</td>
<td>--</td>
<td>GETS</td>
<td>GETS</td>
<td>GETX</td>
</tr>
<tr>
<td></td>
<td>6+64</td>
<td>6+64</td>
<td>6+64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>--</td>
<td>--</td>
<td>GETS</td>
<td>GETS</td>
<td>GETX</td>
</tr>
<tr>
<td></td>
<td>6+64</td>
<td>6+64</td>
<td>6+64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>S</td>
<td>--</td>
<td>--</td>
<td>NA</td>
<td>--</td>
<td>Upgrade</td>
</tr>
<tr>
<td>M</td>
<td>PUTX</td>
<td>PUTX</td>
<td>6 + 64</td>
<td>NA</td>
<td>6 + 64</td>
</tr>
</tbody>
</table>

**Study #1: Bandwidth of MSI vs. MESI**

- X MIPS/MFLOPS processor
  - Use with measured state transition counts to obtain transitions/sec
- Compute state transitions/sec
- Compute bus transactions/sec
- Compute bytes/sec
- What is BW savings of MESI over MSI?
  - MSI requires extra Upgrade for E→M transition
- Result is very small benefit! Why?
  - Small number of E→M transitions (depends on workload!)
  - Upgrade consumes only 6 bytes on bus

**Study #2: MSI Upgrade vs. GETX**

- MSI S→M transition issues Upgrade
  - Could have block invalidated while waiting for Upgrade response
  - Could have block invalidated while waiting for own request on bus!
  - Adds complexity to detect this (and deal with it correctly)
- Could instead just issue GETX
  - Pretend like block is in Invalid state
- Upgrade achieves ~10% to 20% improvement
  - Compared to just issuing GETX
  - Application dependent (as are all studies in architecture!)

**Cache Block Size**

- Block size is unit of transfer and of coherence
  - Doesn’t have to be, could have coherence smaller [Goodman]
- Uniprocessor 3Cs
  - Compulsory, Capacity, Conflict
- Shared memory adds Coherence Miss Type (4th C)
  - True Sharing miss fetches data written by another processor
  - False Sharing miss results from independent data in same coherence block
- Increasing block size
  - Usually fewer 3C misses but more bandwidth
  - Usually more false sharing misses
- P.S. on increasing cache size
  - Usually fewer capacity/conflict misses (& compulsory don’t matter)
  - No effect on true/false “coherence” misses (so may dominate)
Outline

- Motivation for Cache-Coherent Shared Memory
- Snooping Cache Coherence (Chapter 5)
- Implementing Snooping Systems (Chapter 6)
- Advanced Snooping Systems

Implementation Issues

- How does memory know another cache will respond so it doesn’t have to?
- Is it okay if a cache miss is not an atomic event (check tags, queue for bus, get bus, etc.)?
- What about L1/L2 caches & split transactions buses?
- Can we guarantee we won’t get deadlock?
- What happens on a PTE update with multiple TLBs?
- Can one use virtual caches in SMPs?

This is why they pay architects the big bucks!

Outline for Implementing Snooping

- Coherence Control Implementation
- Writebacks, Non-Atomicity
- Hierarchical Caches
- Split Buses
- Deadlock, Livelock, & Starvation
- Three Case Studies
- TLB Coherence
- Virtual Cache Issues

Snooping SMP Design Goals

- Goals
  - Correctness
  - High performance
  - Simple hardware (reduced complexity & cost)

- Conflicts between goals
  - High performance ➔ multiple outstanding low-level events
  ➔ more complex interactions
  ➔ more potential correctness bugs
Base Cache Coherence Design

- Single-level write-back cache
- Invalidation protocol
- One outstanding memory request per processor
- Atomic memory bus transactions
  - No interleaving of transactions
- Atomic operations within a process
  - One finishes before next in program order

Now, we’re going to gradually add complexity
- Why? Faster latencies and higher bandwidths!
- But we’ll stick with invalidation protocol (instead of update)

Cache Controllers and Tags

- On a miss in a uniprocessor:
  - Assert request for memory bus
  - Wait for bus grant
  - Drive address and command lines
  - Wait for command to be accepted by relevant device
  - Transfer data

- In snoop-based multiprocessor, cache controller must:
  - Monitor bus and serve processor
    - Can view as two controllers: bus-side, and processor-side
    - With single-level cache: dual tags (not data) or dual-ported tag RAM
    - Synchronize tags on updates
    - Respond to bus transactions when necessary

Reporting Snoop Results: How?

- Collective response from caches must appear on bus
- Wired-OR signals
  - Shared: asserted if any cache has a copy (used for E state)
  - Dirty/Inhibit: asserted if some cache has a dirty copy
  - Don’t need to know which, since it will do what’s necessary
  - Snoop-valid: asserted when OK to check other two signals

- May require priority scheme for cache-to-cache transfers
  - Which cache should supply data when in shared state?
  - Commercial implementations allow memory to provide data

Reporting Snoop Results: When?

- Memory needs to know what, if anything, to do
- Static delay: fixed number of clocks from address appearing on bus
  - Dual tags required to reduce contention with processor
  - Still must be conservative (update both on write: E → M)
  - Pentium Pro, HP servers, Sun Enterprise (pre E-10K)
- Variable delay
  - Memory assumes cache will supply data until all say “sorry”
  - Less conservative, more flexible, more complex
  - Memory can fetch data early and hold (SGI Challenge)
- Immediately: Bit-per-block state in memory
  - HW complexity in commodity main memory system
**Writebacks**

- Must allow processor to proceed on a miss
  - Fetch the block
  - Perform writeback later

- Need writeback buffer
  - Must handle bus transactions in writeback buffer
  - Snoop writeback buffer
  - Must care about the order of reads and writes
  - Affects memory consistency model (yuck – trust me on this for now)

**Base Organization**

**Optimization #1: Non-Atomic State Transitions**

- Operations involve multiple actions
  - Look up cache tags
  - Bus arbitration
  - Check for outstanding writeback
  - Even if bus is atomic, overall set of actions is not
  - Race conditions among multiple operations

- Suppose P1 and P2 attempt to write cached block A
  - Each decides to issue Upgrade to transition from S → M

- Issues
  - Handle requests for other blocks while waiting to acquire bus
  - Must handle requests for this block A

**Non-Atomicity → Transient States**

Two types of states
- Stable (e.g. MOESI)
- Transient or Intermediate

Increases complexity

Note: Details of this figure aren't important
Optimization #2: Multi-level Cache Hierarchies

- How to snoop with multi-level caches?
  - Independent bus snooping at every level?
  - Maintain cache inclusion?

- Requirements for inclusion
  - Data in higher-level is subset of data in lower-level
  - Modified in higher-level ➞ marked modified in lower-level

- Now only need to snoop lowest-level cache
  - If L2 says not present (modified), then not so in L1

- Is inclusion automatically preserved?
  - Replacements: all higher-level misses go to lower level

Violations of Inclusion

- The L1 and L2 may choose to replace different block
  - Differences in reference history
    - Set-associative first-level cache with LRU replacement
  - Split higher-level caches
    - Instr & data blocks go in different caches at L1, but collide in L2
  - What if L2 is set-associative?
    - Differences in block size

- But a common case works automatically
  - L1 direct-mapped, and
  - L1 has fewer sets than L2, and
  - L1 and L2 have same block size

Inclusion: To Be or Not To Be

- Most common inclusion solution
  - Ensure L2 holds superset of L1I and L1D
  - On L2 replacement or coherence request that must source data or invalidate, forward actions to L1 caches
  - Can maintain bits in L2 cache to filter some actions from forwarding

- But inclusion may not be ideal
  - Restricted associativity in unified L2 can limit blocks in split L1s
  - Not that hard to always snoop L1s
  - If L2 isn’t much bigger than L1, then inclusion is wasteful

- Thus, many new designs don’t maintain inclusion
  - Exclusion: no block is in more than any one cache
  - Not Inclusive is Exclusive and Not Exclusive is Inclusive

Optimization #3: Split-transaction (Pipelined) Bus

- Supports multiple simultaneous transactions
  - Higher throughput!!! (perhaps worse latency)

Atomic Transaction Bus

<table>
<thead>
<tr>
<th>Req</th>
<th>Delay</th>
<th>Response</th>
<th>Time</th>
</tr>
</thead>
</table>

Split-transaction Bus
Potential Problems

- Two transactions to same block (conflicting)
  - Mid-transaction snoop hits
  - E.g., in S, going to M, observe OtherGETX

- Buffering requests and responses
  - Need flow control to prevent deadlock

- Ordering of snoop responses
  - When does snoop response appear with respect to data response?

One Solution (like the SGI PowerPath-2)

- NACK (Negative ACKnowledgment) for flow control
  - Snooper can nack a transaction if it can’t buffer it

- Out-of-order responses
  - Snoop results presented with data response

- Disallow multiple concurrent transactions to one line
  - Not necessary, but it can improve designer sanity

Serialization Point in Split Transaction Buses

- Is the bus still the serialization point?
  - Yes! When a request wins the bus, it is serialized (unless nacked)
  - Data and snoop response can show up way later
  - Snoop decisions are made based on what’s been serialized

- Example (allows multiple outstanding to same block)
  - Initially: block B is in invalid in all caches
  - P1 issues GETX for B, waits for bus
  - P2 issues GETX for B, waits for bus
  - P2’s request wins the bus (but no data from memory until later)
  - P1’s request wins the bus … who responds?
  - P2 will respond, since P2 is the owner (even before data arrives?)
  - P2 receives data from memory
  - P2 sends data to P1

A More General Split-transaction Bus Design

- 4 Buses + Flow Control and Snoop Results
  - Command (type of transaction)
  - Address
  - Tag (unique identifier for response)
  - Data (doesn’t require address)

- Forms of coherence transactions
  - GETS, GETX (both are “request + response”)
  - PUTX (“request + data”)
  - Upgrade (“request”)

- Per Processor Request Table Tracks All Transactions
Multi-Level Caches with Split Bus

More on Correctness

- Partial correctness (never wrong): Maintain coherence and consistency
- Full correctness (always right): Prevent:
  - **Deadlock:**
    - All system activity ceases
    - Cycle of resource dependences
  - **Livelock:**
    - No processor makes forward progress
    - Constant on-going transactions at hardware level
    - E.g. simultaneous writes in invalidation-based protocol
  - **Starvation:**
    - Some processors make no forward progress
    - E.g. interleaved memory system with NACK on bank busy

Deadlock, Livelock, Starvation

- **Deadlock:** Can be caused by request-reply protocols
  - When issuing requests, must service incoming transactions
  - E.g., cache awaiting bus grant must snoop & flush blocks
  - Else may not respond to request that will release bus: deadlock

- **Livelock:**
  - Window of vulnerability problem [Kubi et al., MIT]
  - Handling invalidations between obtaining ownership & write
  - Solution: don’t let exclusive ownership be stolen before write

- **Starvation:**
  - Solve by using fair arbitration on bus and FIFO buffers

Deadlock Avoidance

- Responses are never delayed by requests waiting for a response
- Responses are guaranteed to be sunk
- Requests will eventually be serviced since the number of responses is bounded by the number of outstanding requests
- Must classify messages according to deadlock and coherence semantics
  - If type 1 messages (requests) spawn type 2 messages (responses), then type 2 messages can’t be allowed to spawn type 1 messages
  - More generally, must avoid cyclic dependences with messages
  - We will see that directory protocols often have 3 message types
    - Request, ForwardedRequest, Response
Multi-level Caches with Split-Transaction Bus

- General structure uses queues between
  - Bus and L2 cache
  - L2 cache and L1 cache

- Many potential deadlock problems
- Classify all messages to break cyclic dependences
  - Requests only generates responses
  - Responses don’t generate any other messages
- Requestor guarantees space for all responses
- Use separate request and response queues

Outline

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

Why Snooping Doesn’t Scale

- Limitations of snooping
  - Broadcasting uses lots of “bus” bandwidth
  - Snooping every transaction uses lots of controller bandwidth
- Snooping is great for small-medium size machines
  - Largest current snooping system has 128 processors
- Snooping hits bottleneck for large machines
  - Even with tricks like Multicast Snooping

- So how do we overcome this bottleneck?
  - Get rid of protocol that requires:
    - Broadcasting
    - Logical bus

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
- Maintain directory of which nodes have cached copies of the block (directory controller + directory state)
- On a miss, send message to directory
  - Communication assist via cache controller
  - E.g., invalidations of Shared nodes
- Directory waits for protocol actions to finish and then responds to the original request
- Directory is new serialization point (instead of bus)

Directory Example

Centralized Directory

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

Disadvantages
- Bottleneck (1000s of processors…)
- Directory changes with number of nodes

Advantages
- Only send invalidates/updates to those nodes that have copy of block

Flat, Memory-Based Distributed Directory

- Distribute directory among memory modules
- Maintain directory for each memory block
  - Block’s home node = node with directory info for that block
Directory Nomenclature

- Dir,X -- 2 variables (i and X)
- Directory of i pointers (i <= total number of nodes)
- X specifies what to do on Shared to Modified transition
  - B = Broadcast
  - NB = No Broadcast
  - SW = Software
- DirN = full-map directory
  - Bit vector per memory block
  - Bit per node in system
  - No need to broadcast

Limited Pointer Directory

- Each directory entry contains < N pointers
  - Not bit vector, but instead pointers to nodes
- Less overhead than full-map directory
- What to do when run out of pointers depends on what we want to do on S→M transition

Broadcast (Dir,B)
- Just give out another copy of block
- Modify state to indicate broadcast
- If S→M, then broadcast invalidation

No Broadcast (Dir,NB)
- Never allow more than i Sharers
  - If another request for shared, then invalidate a current sharer

Replacement Notification

- Should the directory be notified for blocks that are replaced from Shared state?

Reasons to do this
- Can avoid broadcast, clear bit/pointer when notified

Reasons not to do this
- Read-only data that will never be invalidated
- Notifications cause unnecessary traffic

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 Directory (aka Directory Caching)
- Not all of memory is in processor caches
  - Size of memory >> sum of cache sizes
- Cache of directory entries at memory
**Software Assistance**

- Trap to software if we run out of pointers
- Limitless Directory (MIT Alewife)
- Dir,SW (Wisconsin Wind Tunnel Group)

Why software assistance?
- Cost (less hardware)
- Flexibility

Actually, can do *everything* in software = Software DSM
- Page-based DSM
- Blizzard-S
- Shasta

**Flat, Cache-Based Distributed Directory (SCI)**

- Also known as “chaining directory”
- Build linked list of nodes containing cache block
- Store pointers in cache with block of data
- Home node points to start of list

![Diagram of a linked list in SCI]

**Chaining (SCI)**

- Scalable Coherent Interface (IEEE standard)
- Build linked list of nodes containing cache block
- Store pointers in cache with block of data
- Home node points to start of list

Issues
- Replacements
- Time to send invalidations

**End Coherence Overview**

- Snooping coherence protocols
- Implementation Issues
- Directory overview
- Remaining slides are for your enjoyment, please read over them.
SGI Challenge Overview

- 36 MIPS R4400 (peak 2.7 GFLOPS, 4 per board) or 18 MIPS R8000 (peak 5.4 GFLOPS, 2 per board)
- 8-way interleaved memory (up to 16 GB)
- 1.2 GB/s Powerpath-2 bus @ 47.6 MHz, 16 slots, 329 signals
- 128-Byte lines (1 + 4 cycles)
- Split-transaction with up to 8 outstanding reads
  - All transactions take five cycles
- Miss latency nearly 1 us (mostly on CPU board, not bus...)

Invalidate vs. Update, cont.

- Pattern 1 (one write before reads)
  - N = 16, M = 10, K = 10
  - Update
    - Iteration 1: N regular cache misses (70 bytes)
    - Remaining iterations: update per iteration (14 bytes; 6 cntrl, 8 data)
  - Total Update Traffic = 16*70 + 9*14 = 1,246 bytes
  - Book assumes 10 updates instead of 9...
  - Invalidate
    - Iteration 1: N regular cache misses (70 bytes)
    - Remaining: P1 generates upgrade (6), 15 others Read miss (70)
  - Total Invalidate Traffic = 16*70 + 9*6 + 15*9*17 = 10,624 bytes

- Pattern 2 (many writes before reads)
  - Update = 1,400 bytes
  - Invalidate = 824 bytes

What about real workloads?

- Update can generate too much traffic
- Must selectively limit it

Current assessment

- Update very hard to implement correctly (because of consistency ... discussion coming in a couple weeks)
- Rarely done

Future assessment

- May be same as current or
- Chip multiprocessors may revive update protocols
  - More intra-chip bandwidth
  - Easier to have predictable timing paths?

Study #3: Invalidate vs. Update

- Pattern 1:
  for i = 1 to k
  P1(write, x);         // one write before reads
  P2 to PN-1(read, x);
end for i

- Pattern 2:
  for i = 1 to k
  for j = 1 to m
  P1(write, x);        // many writes before reads
  P2(read, x);
end for j
end for i
- 4 MIPS R4400 processors per board share A / D chips
- A chip has address bus interface, request table, control logic
- CC chip per processor has duplicate set of tags
- Processor requests go from CC chip to A chip to bus
- 4 bit-sliced D chips interface CC chip to bus

**SGI Powerpath-2 Bus**

- Non-multiplexed (i.e., separate A and D), 256-data/40-address, 47.6 MHz, 8 outstanding requests
- Wide \( \Rightarrow \) more interface chips so higher latency, but more bandwidth at slower clock
- Large block size also calls for wider bus
- Uses Illinois MESI protocol (cache-to-cache sharing)
- More detail in chapter

**Bus Design and Request-Response Matching**

- Essentially two separate buses, arbitrated independently
  - "Request" bus for command and address
  - "Response" bus for data
- Out-of-order responses imply need for matching request with corresponding response
  - Request gets 3-bit tag when wins arbitration (8 outstanding max)
  - Response includes data as well as corresponding request tag
  - Tags allow response to not use address bus, leaving it free
- Separate bus lines for arbitration and for snoop results

**Bus Design (continued)**

- Each of request and response phase is 5 bus cycles
  - Response: 4 cycles for data (128 bytes, 256-bit bus), 1 turnaround
  - Request phase: arbitration, resolution, address, decode, ack
  - Request-response transaction takes 3 or more of these
- Determine who will respond, if any
- Actual response comes later, with re-arbitration
- Write-backs have request phase only; arbitrate both data+addr buses

Cache tags looked up in decode; extend ack cycle if not possible
Bus Design (continued)

- Flow-control through negative acknowledgement (NACK)
- No conflicting requests for same block allowed on bus
  - 8 outstanding requests total, makes conflict detection tractable
  - Eight-entry "request table" in each cache controller
  - New request on bus added to all at same index, determined by tag
  - Entry holds address, request type, state in that cache (if determined already), ...
  - All entries checked on bus or processor accesses for match, so fully associative
  - Entry freed when response appears, so tag can be reassigned by bus

Bus Interface with Request Table

Memory Access Latency

- 250ns access time from address on bus to data on bus
- But overall latency seen by processor is 1000ns!
  - 300 ns for request to get from processor to bus
  - Down through cache hierarchy, CC chip and A chip
  - 400ns later, data gets to D chips
  - 3 bus cycles to address phase of request transaction, 12 to access main memory, 5 to deliver data across bus to D chips
  - 300ns more for data to get to processor chip
  - Up through D chips, CC chip, and 64-bit wide interface to processor chip, load data into primary cache, restart pipeline

Challenge I/O Subsystem

- Multiple I/O cards on sys bus, each w/320MB/s HIO bus
  - Personality ASIs connect these to devices (standard and graphics)
- Proprietary HIO bus
  - 64-bit multiplexed address/data, split read trans., up to 4 per device
  - Pipelined, but centralized arbitration, with several transaction lengths
  - Address translation via mapping RAM in system bus interface
- I/O board acts like a processor to memory system
**SUN Enterprise 6000 Overview**

- Up to 30 UltraSPARC processors, MOESI protocol
- Gigaplane™ bus has peak bw 2.67 GB/s, 300 ns latency
- Up to 112 outstanding transactions (max 7 per board)
  - 16 bus slots, for processing or I/O boards
    - 2 CPUs and 1GB memory per board
    - Memory distributed, but protocol treats as centralized (UMA)

**Sun Gigaplane Bus**

- Non-multiplexed, split-transaction, 256-data/41-address, 63.5 MHz (plus 32 ECC lines, 7 tag, 18 arbitration, etc. Total 388)
- Cards plug in on both sides: 8 per side
- 112 outstanding transactions, up to 7 from each board
- Designed for multiple outstanding transactions per processor
- Emphasis on reducing latency, unlike Challenge
  - Speculative arbitration if address bus not scheduled from prev. cycle
  - Else regular 1-cycle arbitration, and 7-bit tag assigned in next cycle
- Snoop result associated with request (5 cycles later)
- Main memory can stake claim to data bus 3 cycles into this, and start memory access speculatively
- Two cycles later, asserts tag bus to inform others of coming transfer
- MOESI protocol

**Enterprise Processor and Memory System**

- 2 procs / board, ext. L2 caches, 2 mem banks w/ x-bar
- Data lines buffered through UDB to drive internal 1.3 GB/s UPA bus
- Wide path to memory so full 64-byte line in 2 bus cycles

**Enterprise I/O System**

- I/O board has same bus interface ASICs as processor boards
- But internal bus half as wide, and no memory path
- Only cache block sized transactions, like processing boards
  - Uniformity simplifies design
  - ASICs implement single-block cache, follows coherence protocol
- Two independent 64-bit, 25 MHz Sbuses
  - One for two dedicated FiberChannel modules connected to disk
  - One for Ethernet and fast wide SCSI
  - Can also support three SBUS interface cards for arbitrary peripherals
- Performance and cost of I/O scale with # of I/O boards
**Sun Enterprise 10000 (aka E10K or Starfire)**

- How far can you go with snooping coherence?
- Quadruple request/snoop bandwidth using four “logical” address buses
  - Each handles 1/4 of physical address space
  - Impose logical ordering for consistency: for writes on same cycle, those on bus 0 occur “before” bus 1, etc.
- Get rid of data bandwidth problem: use a network
  - E10k uses 16x16 crossbar between CPU boards & memory boards
  - Each CPU board has up to 4 CPUs: max 64 CPUs total
- 10.7 GB/s max BW, 468 ns unloaded miss latency
- We will discuss a paper on E10K later

---

**Outline for Implementing Snooping**

- Coherence Control Implementation
- Writebacks, Non-Atomicity, & Serialization/Order
- Hierarchical Cache
- Split Buses
- Deadlock, Livelock, & Starvation
- Three Case Studies
- TLB Coherence
- Virtual Cache Issues

---

**Translation Lookaside Buffer**

- Cache of page table entries
- Page table maps virtual page to physical frame

**The TLB Coherence Problem**

- Since TLB is a cache, must be kept coherent
- Change of PTE on one processor must be seen by all processors
- Why might a PTE be cached in more than 1 TLB?
  - Actual sharing of data
  - Process migration
- Changes are infrequent
  - Get OS to do it
TLB Shootdown

• To modify TLB entry, modifying processor must
  – LOCK page table,
  – Flush TLB entries,
  – Queue TLB operations,
  – Send inter-processor interrupt,
  – Spin until other processors are done
  – UNLOCK page table

• SLOW!

• But most common solution today

TLB Shootdown Improvements

• Evolutionary Changes
  – Keep track of which processor even had the mapping
    & only shoot them down
  – Defer shootdowns on “upgrade” changes
    (e.g., page from read-only to read-write)
  – SGI Origin “poison” bit for also deferring downgrades

• Revolutionary Changes
  – “Invalidate TLB entry” instruction (e.g., PowerPC)
  – No TLB (e.g., Berkeley SPUR)
    » Use virtual L1 caches so address translation only on miss
    » On miss, walk PTE (which will often be cached normally)
    » PTE changes kept coherent by normal cache coherence

Virtual Caches & Synonyms

• Problem
  – Synonyms: V0 & V1 map to P1
  – When doing coherence on block in P1, how do you find V0 & V1?

• Don’t do virtual caches (most common today)

• Don’t allow synonyms
  – Probably use a segmented global address space
  – E.g., Berkeley SPUR had process pick 4 of 256 1GB segments
  – Still requires reverse address translation

• Allow virtual cache & synonyms
  – How do we implement reverse address translation?
    – See Wang et al. next

Wang et al. [ISCA89]

• Basic Idea
  – Extended Goodman one-level cache idea [ASPLOS87]
  – Virtual L1 and physical L2
  – Do coherence on physical addresses
  – Each L2 block maintains pointer to corresponding L1 block (if any)
    (requires log2 #L1_blocks - log2 (page_size / block_size)
  – Never allow block to be simultaneously cached under synonyms

• Example where V0 & V1 map to P2
  – Initially V1 in L1 and P2 in L2 points to V1
  – Processor references V0
  – L1 miss
  – L2 detects synonym in L1
  – Change L1 tag and L2 pointer so that L1 has V0 instead of V1
  – Resume
Virtual Caches & Homonyms

- Homonym
  - “Pool” of water and “pool” the game
  - V0 of one process maps to P2, while V0 of other process maps to P3

- Flush cache on context switch
  - Simple but performs poorly

- Address-space IDs (ASIDs)
  - In architecture & part of context state

- Mapping-valid bit of Wang et al.
  - Add mapping-valid as a “second” valid bit on L1 cache block
  - On context switch do “flash clear” of mapping-valid bits
  - Interesting case is valid block with mapping invalid
    - On processor access, re-validate mapping
    - On replacement (i.e., writeback) treat as valid block

Outline for Implementing Snooping

- Coherence Control Implementation
- Writebacks, Non-Atomicity, & Serialization/Order
- Hierarchical Cache
- Split Buses
- Deadlock, Livelock, & Starvation
- Three Case Studies
- TLB Coherence
- Virtual Cache Issues

Outline

- Motivation for Cache-Coherent Shared Memory
- Snooping Cache Coherence
- Implementing Snooping Systems
- Advanced Snooping Systems
  - Sun UltraEnterprise 10000
  - Multicast Snooping (Wisconsin)

Sun UltraEnterprise 10000 (Starfire)

- Shared-wire bus is bottleneck in snooping systems
  - Tough to implement at high speed
  - Centralized shared resource
- Solution: multiple “logical buses”
- PRESENTATION
Multicast Snooping

- Bus is bottleneck in snooping systems
  - But why broadcast requests when we can multicast?

- PRESENTATION

Outline

- Motivation for Cache-Coherent Shared Memory
- Snooping Cache Coherence
- Implementing Snooping Systems
- Advanced Snooping Systems