Review: Symmetric Multiprocessors (SMP)

- Multiple (micro-)processors
- Each has cache (today a cache hierarchy)
- Connect with logical bus (totally-ordered broadcast)
- Implement Snooping Cache Coherence Protocol
  - Broadcast all cache “misses” on bus
  - All caches “snoop” bus and may act
  - Memory responds otherwise
Review: Snoopy Design Choices

- Controller updates state of blocks in response to processor and snoop events and generates bus xactions
- 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

(C) 2005 Mark D. Hill from Adve, Falsafi, Lebeck, Reinhard, Singh, & Smith
But in More Detail ...

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

Outline

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

• **Goals**
  - Correctness
  - High Performance
  - Minimal Hardware => reduced complexity & cost

• **Often at odds**
  - 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 process
  – one finishes before next in program order
• Examine write serialization, completion, atomicity
• Then add more concurrency and re-examine
Cache Controller and Tags

• On a miss in uniprocessor:
  – Assert request for 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 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 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
  – Dirty/Inhibit: asserted if some cache has a dirty copy
    » needn’t 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
- Fixed number of clocks from address 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
- Variable delay
  - Memory assumes cache will supply data till all say "sorry"
  - Less conservative, more flexible, more complex
  - Memory can fetch data early and hold (SGI Challenge)
- Immediately: Bit-per-block in memory
  - H/W complexity in commodity main memory system

Writebacks

- Must allow processor to proceed on a miss
  - fetch the block
  - perform writeback later
- Need writebuffer
  - Must handle bus transactions in write buffer
  - Snoop writebuffer
  - Must care about the order of reads and writes
  - Revisit in Adve's tutorial
Non-Atomic State Transitions

- Operations involve multiple actions
  - Look up cache tags
  - Bus arbitration
  - Check for 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 BusUpgr to allow 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. MESI)
• Transient or Intermediate

Increases complexity

Serialization and Ordering

• Let A and flag be 0
• P1
• A += 5 while (flag == 0)
• flag = 1 print A

• Assume A and flag are in different cache blocks
• What happens?
• How do you implement it correctly?
Serialization and Ordering

- Processor-cache handshake must preserve serialization
- e.g. write to S state=> first obtain ownership
- why?
- Write completion for SC => need bus invalidation:
  - Wait to get bus, can proceed afterwards
- Must serialize bus operations in program order

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
  - Modifications
Violations of Inclusion

• The two caches (L1, L2) may choose to replace different block
  
  Example: Local LRU not sufficient
  
  Assume that L1 and L2 hold two and three blocks and both
  use local LRU
  
  Processor references: 1, 2, 1, 3, 1, 4
  
  Final contents of L1: 1, 4
  
  L1 misses: 1, 2, 3, 4
  
  Final contents of L2: 2, 3, 4, but not 1

Violations of Inclusion

• Split higher-level caches
  
  – instruction, data blocks go in different caches at L1, but collide in L2

• Differences in Associativity
  
  – What if L1 is set-associative and L2 is direct-mapped?

• Differences in block size
  
  – Blocks in two L1 sets may both map to same L2 set

• But a common case works automatically
  
  – L1 direct-mapped, fewer sets than in L2, and block size same
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
    - virtual L1 / physical [Wang, et al., ASPLOS87]
  - But
    - Restricted associativity in unified L2 can limit blocks in split L1’s
    - “Backside” L2 (bus-L1-processor-L2) makes filtering awkward
    - Not that hard to always snoop L1’s

- Thus, many new designs don’t maintain inclusion

Shared Caches

- Share low level caches among multiple processors
  - Sharing L1 adds to latency, unless multithreaded processor

- Advantages
  - Eliminates need for coherence protocol at shared level
  - Reduces latency within sharing group
  - Processors essentially prefetch for each other
  - Can exploit working set sharing
  - Increases utilization of cache hardware

- Disadvantages
  - Higher bandwidth requirements
  - Increased hit latency
  - May be more complex design
  - Lower effective capacity if working sets don’t overlap

- Bottom Line
  - Packaging has alot to do with it
  - As levels of integrations increase, there will be more sharing
**Split-transaction (Pipelined) Bus**

- Supports multiple simultaneous transactions (many designs)

**Atomic Transaction Bus**

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

**Split-transaction Bus**

- Two transactions to same block (conflicting)
  - Mid-transaction snoop hits
  - Disallow (SGI Challenge)
  - Logically Pipeline (Sun Gigaplane)

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

- Ordering of Snoop responses
  - When does snoop response appear wrt data response
Multi-Level Caches with Split Bus

- General structure uses queues between
  - Bus and L2 cache
  - L2 cache and L1 cache
- Deadlock!
- Classify all transactions
  - Request, only generates responses
  - Response, doesn’t generate any other transactions
- Requestor guarantees space for all responses
- Use Separate Request and Response queues
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

• Request-reply protocols can lead to deadlock
  – 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 outstanding requests
- Must classify transactions according to deadlock and coherence semantics
  - e.g., ordering of BusRD response (Bdata) and BInval
  - Treat both Bdata and Binval as requests (go in same queue)

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, 83.5 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

Gigaplane Bus Timing

<table>
<thead>
<tr>
<th>Address</th>
<th>Rd A Tag</th>
<th>Rd B Tag</th>
<th>State</th>
<th>Arbitration</th>
<th>Tag</th>
<th>Status</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>4,5</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>Share</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td>Own</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td>Cancel</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(C) 2005 Mark D. Hill from Adve, Falsafi, Lebeck, Reinhardt, Singh, & Smith
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 no. of I/O boards
Memory Access Latency

- 300 ns read miss latency (130 ns on bus)
- Rest is path through caches & the DRAM access
- TLB misses add 340 ns

Ping-pong microbenchmark is 1.7 μs round-trip (5 mem accesses)

Sun Enterprise 10000

- How far can you go with snooping coherence?
- Quadruple request/snoop bandwidth using four address busses
  - 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
  - E10000 uses 16x16 crossbar betw. 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
- See “Starfire: Extending the SMP Envelope”, IEEE Micro, Jan/Feb 1998
Outline

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

Translation Lookaside Buffer

• Cache of Page Table Entries
• Page Table Maps Virtual Page to Physical Frame

Virtual Address Space

| 0 | 4 | 7 |

Physical Address Space

| 3 | 4 | 7 |
**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
- Process migration
- Changes are infrequent
  - get OS to do it
  - Always flush TLB is often adequate

**TLB Shootdown**

- To modify TLB entry, modifying processor must
  - LOCK page table,
  - flush TLB entries,
  - queue TLB operations,
  - send interprocessor interrupt,
  - spin until other processors are done
  - UNLOCK page table

- SLOW...
  - But most common solution today
- Some ISAs have “flush TLB entry” instructions
**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**
  - Constrains software (and OS assumptions)
- **Allow virtual cache & synonyms**
  - How implement reverse address translation?
  - See Wang et al. next

---

**Wang et al. [ISCA89]**

- **Basic Idea**
  - Virtual L1 and physical L2
  - Do coherence on physical addresses
  - Each L2 block maintains backpointer to corresponding L1 block (if any)
    (requires $\log_2 \#L1\_blocks - \log_2 (\text{page\_size} / \text{block\_size})$
  - Never allow block to be simultaneously cached under synonyms
- **Example where V0 & V1 map to P2**
  - Initially V1 in L1 and P2 in L1 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**
  - 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

Outline

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