Outline

• Motivation for Cache-Coherent Shared Memory
• Snooping Cache Coherence
• Implementing Snooping Systems
• Advanced Snooping Systems
Review: Symmetric Multiprocessors (SMP)

- Multiple (micro-)processor cores
- 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
Note: we never take any action on an OtherPUTX
Some (but not all!) 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 last-level 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 core 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

(C) 2010 Daniel J. Sorin from Adve, Falsafi, Hill, Lebeck, Reinhardt, Singh

ECE 259 / CPS 221
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 $\rightarrow$ Transient States

Two types of states

- Stable (e.g. MOESI)
- Transient or Intermediate

Increases complexity

In-class exercise: let’s figure out how many states we really need in an “MSI” protocol …
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 != Exclusive and Not Exclusive != Inclusive
Optimization #3: Split-transaction (Pipelined) Bus

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

Atomic Transaction Bus

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

Processor

L$_1$ $\uparrow$

Response

L$_2$ $\downarrow$

Response/ request from L$_2$ to L$_1$

L$_1$ $\uparrow$

Response/ request from L$_1$ to L$_2$

L$_2$ $\downarrow$

Response/ request from bus

Request/response to bus

Bus

(C) 2010 Daniel J. Sorin from Adve, Falsafi, Hill, Lebeck, Reinhardt, Singh
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
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