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 actions
- 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) 2001 Mark D. Hill from Adve, Falsafi, Lebeck, Reinhardt, & Singh
CS/ECE 757
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
- Three Case Studies
- TLB Coherence
- Virtual Cache Issues
Snooping SMP Design Goals

• Goals
  – Correctness
  – High Performance
  – Minimal Hardware $\Rightarrow$ reduced complexity & cost

• Often at odds
  – High Performance
    $\Rightarrow$ multiple outstanding low-level events
    $\Rightarrow$ more complex interactions
    $\Rightarrow$ 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 or pipelining 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: asserted if some cache has a dirty copy
    » Needn’t know which, since it will do what’s necessary
  – Inhibit: asserted until OK to check other two signals

• May require priority scheme for cache-to-cache transfers
  – Which cache should supply data when in shared state?
  – Memory supplies data when in shared state in many commercial implementations
Reporting Snoop Results: When?

- Memory needs to know what, if anything, to do
  
- Fixed number of clocks from address appearing on bus
  - Dual tags required to reduce contention with processor
  - Timing must be conservative (update on E -> M may busy tags)
  - Pentium Pro (w/ inhibit), 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 miss block immediately
  - perform writeback later

- Need writeback buffer
  - Must handle bus transactions for data in writeback buffer
  - Snoop writeback buffer
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 \rightarrow M
  - How is this conflict resolved?

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

- Solution: Add Transient States
### Non-Atomicity => Transient States

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

Increases complexity

---

### Serialization and Ordering

- **An optimization:** If A is shared, make upgrade request, then write A in local cache, and continue.
  - i.e. don’t wait for bus transaction
- **What can happen?**
- **How do you implement it correctly?**
Serialization and Ordering

- Processor-cache handshake must preserve serialization
  e.g. write to S state $\Rightarrow$ first obtain ownership
  why?
- Write completion for SC $\Rightarrow$ need bus invalidation:
  - Wait to get bus, can proceed afterwards;
  - i.e. commit occurs when bus is acquired
- 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: low level acts as snoop “filter”
- Requirements for Inclusion
  - data in higher-level is subset of data in lower-level
  - modified in higher-level $\Rightarrow$ marked modified in lower-level
- Now only need to snoop lowest-level cache
  - If L2 says not present (modified), then same holds for L1
- To Preserve Inclusion:
  - Replacements: all higher-level misses go to lower level
  - Changes from bus: must be forwarded to higher levels
  - Modifications at L1: must be propagated to lower levels
Violations of Inclusion

- The two caches (L1, L2) may choose to replace different blocks
  
  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

- 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 inclusion bits in L2 cache to filter some actions from forwarding

- **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, some 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
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 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 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
Split-transaction (Pipelined) Bus

- Supports multiple simultaneous transactions (many designs)

Atomic Transaction Bus

| Req | Delay | Response |

Split-transcation Bus

| Request | Delay | Response |

Split-Transaction Potential Problems

- Bus becomes non-atomic
- Two ongoing transactions to same block (conflicting)
  - Mid-transaction snoop hits
- Buffer requests and responses
  - Need flow control to prevent deadlock
- Ordering of Snoop responses
  - when does snoop response appear wrt data response
One Solution

- Disallow conflicting transactions
- Request Phase establishes serialization order
- NACK for flow control
- Out-of-order responses
  - to allow for variable memory latencies
  - snoop results presented with data response

A Split-transaction Bus Design

- 2 Buses (Address/Data)
  - Address bus:
    » address, command, tag (unique identifier for transaction)
  - Data bus:
    » tag, data
- Form of transactions
  - BusRD, BusRXD (request + response)
  - Writeback (request + data)
  - Upgrade (request only)
- Per Processor Request Table Tracks All Transactions
A Simple Example

P2 Can snoop data from first ld
P1 Must hold st operation until entry is clear

Multi-Level Caches with Split Bus

(C) 2001 Mark D. Hill from Adve,
Falsafi, Lebeck, Reinhardt, & Singh  CS/ECE 757
Multi-level Caches with Split-Transaction Bus

- General structure uses queues between
  - Bus and L2 cache
  - L2 cache and L1 cache
- Can deadlock!
- Say L1 to L2 queue holds read request which will cause an L2 response to L1
- Say L2 to L1 queue holds an invalidate request which will cause an L1 response to L2 (dirty block)

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

• 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
• 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)