18-742

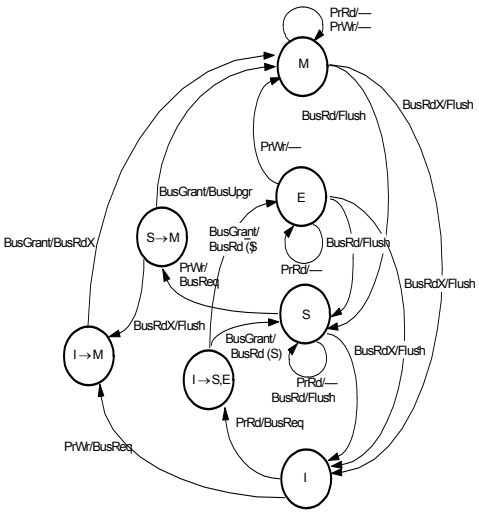
Lecture 9

Symmetric Multiprocessors II

Spring 2005

Prof. Babak Falsafi

http://www.ece.cmu.edu/~ece742



Slides developed in part by Profs. Adve, Falsafi, Hill, Lebeck, Reinhardt, Smith, and Singh of University of Illinois, Carnegie Mellon University, University of Wisconsin, Duke University, University of Michigan, and Princeton University.

Announcements & Readings

Homework 5 will be posted today

Talk at Intel this Friday

Can Parallel Computing Finally Impact Mainstream Computing?

Uzi Vishkin, University of Maryland

(10:30am @ Intel Research Pittsburgh)

Chapter 6 of the book

Alan Charlesworth, *StarFire: Extending the SMP Envelope*, IEEE Micro, Jan. 1998.

Implementing a Centralized Barrier

```
BARRIER(bar_name, p) {
    LOCK(bar_name.lock);
    if (bar_name.counter = 0)
      bar_name.flag = 0;
    bar_name.counter++;
    UNLOCK(bar_name.lock);
    if (bar_name.counter == p) {
      bar_name.counter = 0;
      bar_name.flag = 1;
    }
    else
      while(bar_name.flag = 0) {};        /* busy wait */
  }
```

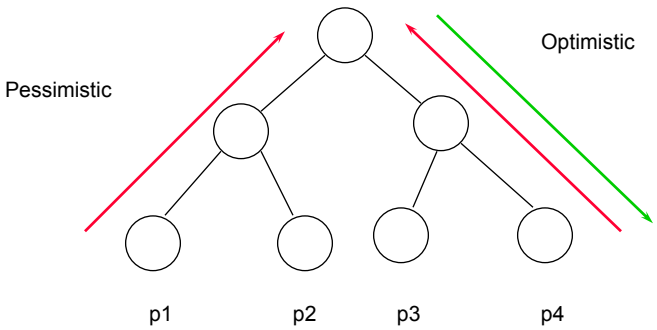
- **Does this work?**

Barrier With Sense Reversal

```
BARRIER(bar_name, p) {
    local_sense = !(local_sense);              /* toggle private state */
    LOCK(bar_name.lock);
    bar_name.counter++;
    UNLOCK(bar_name.lock);
    if (bar_name.counter == p) {
      bar_name.counter = 0;
      bar_name.flag = local_sense;
    }
    else
      while(bar_name.flag != local_sense) {};    /* busy wait */
  }
```

Synchronization Algorithms

- **Tournament Barriers, SW Combining Tree**

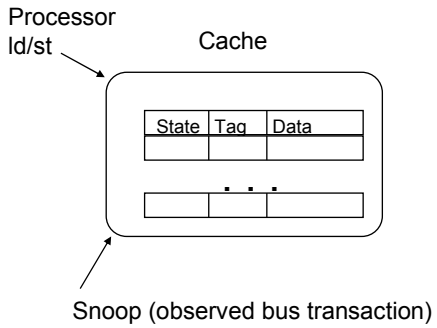


Review: Symmetric Multiprocesors (SMP)

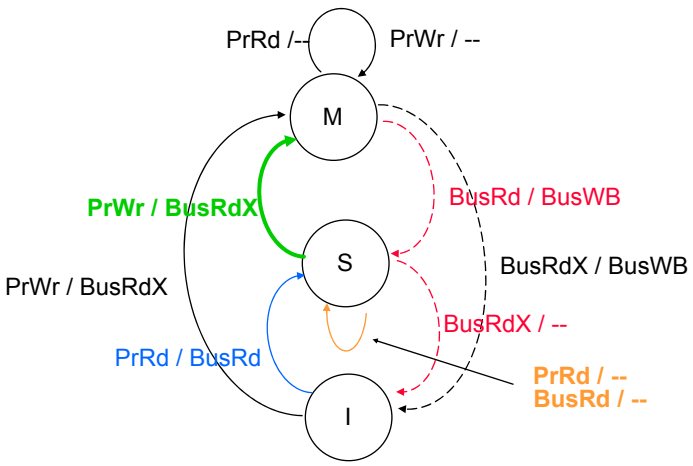
- **Multiple (micro-)processors**
- **Each has cache (today a cache hierarchy)**
- **Connect with logical bus (totally-ordered broadcast)**
- **Implement Snooping Cache Coherence Protocol**
  - **Broadcast all cache "misses" on bus**
  - **All caches "snoop" bus and may act**
  - **Memory responds otherwise**

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



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 => 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

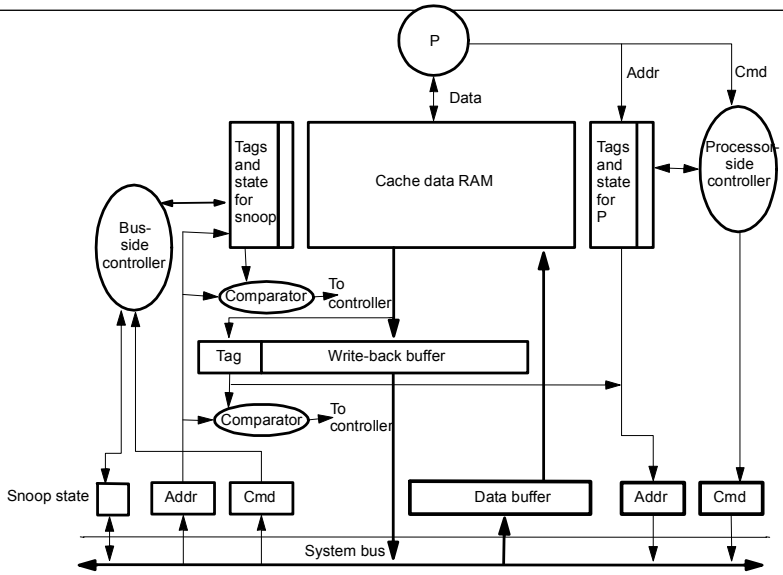
Reporting Snoop Results: When?

- Memory needs to know what, if anything, to do
- Fixed number of clocks from address 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
- 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

Base Organization



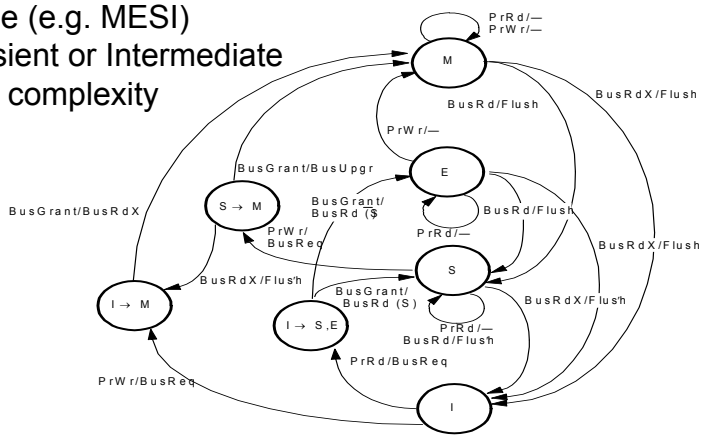
Non-Atomic State Transitions

- **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	**P2**
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**

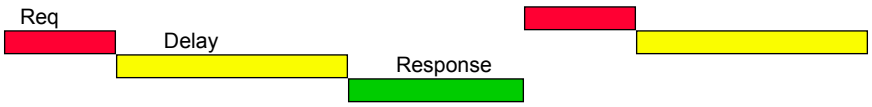
Shared Caches

- **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 a lot 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



Split-transcation Bus

Potential Problems

- **Two 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**

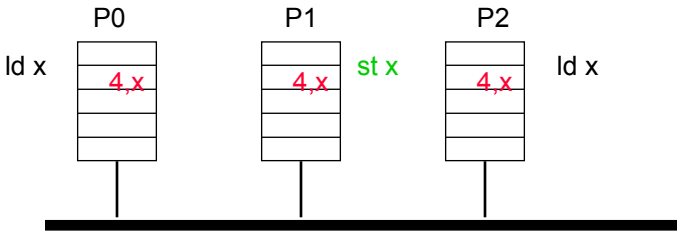
One Solution

- NACK for flow control
- Out-of-order responses
  - snoop results presented with data response
- Disallow conflicting transactions

A Split-transaction Bus Design

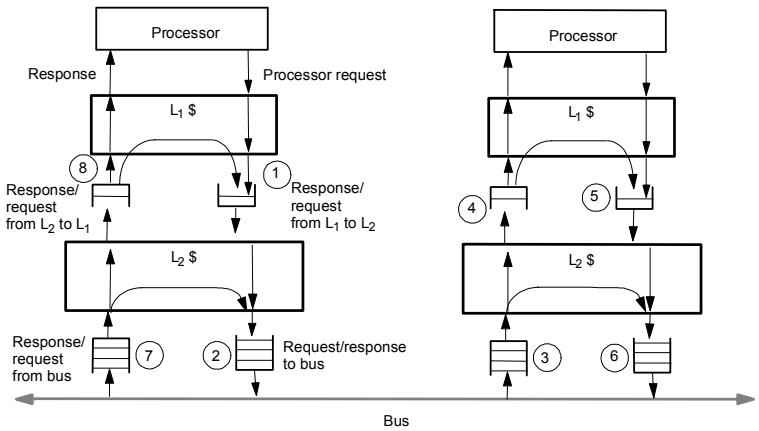
- 4 Buses + Flow Control and Snoop Results
  - Command (type of xaction)
  - Address
  - Tag (unique identifier for response)
  - Data (doesn't require address)
- Form of transactions
  - BusRD, BusRDX (request + response)
  - Writeback (request + data)
  - Upgrade (request only)
- Per Processor Request 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



Multi-level Caches with Split-Transaction 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