CS/ECE 757: Advanced Computer Architecture II
(Parallel Computer Architecture)

Symmetric Multiprocessors Part 1 (Chapter 5)

Copyright 2001 Mark D. Hill
University of Wisconsin-Madison

Slides are derived from work by
Sarita Adve (Illinois), Babak Falsafi (CMU),
Alvy Lebeck (Duke), Steve Reinhardt (Michigan),
and J. P. Singh (Princeton). Thanks!

Outline

• Motivation
• Coherence
• Coherence Tradeoffs
• Memory Consistency
• Synchronization
What is (Hardware) Shared Memory?

- Take multiple (micro-)processors
- Implement a memory system with a single global physical address space (usually)
- Minimize memory latency (co-location & caches)
- Maximize memory bandwidth (parallelism & caches)

Some Memory System Options

(a) Shared cache
(b) Bus-based shared memory
(c) Dancehall
(d) Distributed memory
Why Shared Memory?

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

• Minuses
  – Proper synchronization is complex
  – Communication is implicit so harder to optimize
  – Hardware designers must implement

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

In More Detail

• Efficient Naming
  – virtual to physical using TLBs
  – ability to name relevant portions of objects

• Ease and efficiency of caching
  – caching is natural and well understood
  – can be done in HW automatically

• Communication Overhead
  – low since protection is built into memory system
  – easy for HW to packetize requests / replies

• Integration of latency tolerance
  – demand-driven: consistency models, prefetching, multithreaded
  – Can extend to push data to PEs and use bulk transfer
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

Cache Coherence Problem (Step 1)

(C) 2001 Mark D. Hill from Adve, Falsafi, Lebeck, Reinhart, & Singh
CS/ECE 757
Cache Coherence Problem (Step 2)

P1

P2

Id r2, x

Id r2, x

Interconnection Network

Interconnection Network

Main Memory

Main Memory

Cache Coherence Problem (Step 3)

P1

P2

Id r2, x

Id r2, x

Interconnection Network

Interconnection Network

X =========

X =========

Main Memory

Main Memory

(C) 2001 Mark D. Hill from Adve, Falsafi, Lebeck, Reinhardt, & Singh
Snoopy Cache-Coherence Protocols

- Bus provides serialization point (more on this later)

- Each cache controller “snoops” all bus transactions
  - relevant transactions if for a block it contains
  - take action to ensure coherence
    » invalidate
    » update
    » supply value
  - depends on state of the block and the protocol

- Simultaneous Operation of Independent Controllers

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
The Simple Invalidate Snooping Protocol

- Write-through, no-write-allocate cache
- Actions: PrRd, PrWr, BusRd, BusWr

### A 3-State Write-Back Invalidation Protocol

- **2-State Protocol**
  - Simple hardware and protocol
    - Bandwidth (every write goes on bus!)
- **3-State Protocol (MSI)**
  - Modified
    - one cache has valid/latest copy
    - memory is stale
  - Shared
    - one or more caches have valid copy
  - Invalid
- Must invalidate all other copies before entering modified state
- Requires bus transaction (order and invalidate)
**MSI Processor and Bus Actions**

- **Processor:**
  - PrRd
  - PrWr
  - Writeback on replacement of modified block

- **Bus**
  - Bus Read (BusRd) Read without intent to modify, data could come from memory or another cache
  - Bus Read-Exclusive (BusRdX) Read with intent to modify, must invalidate all other caches copies
  - Writeback (BusWB) cache controller puts contents on bus and memory is updated
  - Definition: cache-to-cache transfer occurs when another cache satisfies BusRd or BusRdX request

- **Let’s draw it!**

---

**MSI State Diagram**

![MSI State Diagram](image)
An 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>1. P1 read u</td>
<td>S</td>
<td>--</td>
<td>--</td>
<td>BusRd</td>
<td>Memory</td>
</tr>
<tr>
<td>2. P3 read u</td>
<td>S</td>
<td>--</td>
<td>S</td>
<td>BusRd</td>
<td>Memory</td>
</tr>
<tr>
<td>3. P3 write u</td>
<td>I</td>
<td>--</td>
<td>M</td>
<td>BusRdX</td>
<td>Memory or not</td>
</tr>
<tr>
<td>4. P1 read u</td>
<td>S</td>
<td>--</td>
<td>S</td>
<td>BusRd</td>
<td>P3's cache</td>
</tr>
<tr>
<td>5. P2 read u</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>BusRd</td>
<td>Memory</td>
</tr>
</tbody>
</table>

- Single writer, multiple reader protocol
- Why Modified to Shared?
- What if not in any cache?
  - Read, Write produces 2 bus transactions!

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
- Cache Flush for cache-to-cache transfers
  - Only one can do it though
4-State (MESI) Invalidation Protocol

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
  - MOSI
  - 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)
- **Adds Bus Update Transaction**
- **Adds Cache Controller Update operation**
- **Must obtain bus before updating local copy**
- **What does state diagram look like?**
  - let’s look at the actions first

---

**Dragon Actions**

- **Processor**
  - PrRd
  - PrWr
  - PrRdMiss
  - PrWrMiss
  - Update in response to BusUpd
- **Bus Xactions**
  - BusRd
  - BusUpd
  - BusWB
Tradeoffs in Protocol Design

- New State Transitions
- What Bus Transactions
- Cache block size
- Workload dependence
- Compute bandwidth, miss rates, from state transitions
Computing Bandwidth

- Why bandwidth?
- How do I compute it?
- Monitor State Transitions
  - tells me 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>BusRd 6+64</td>
<td>BusRd 6+64</td>
<td>BusRdX 6+64</td>
</tr>
<tr>
<td>I</td>
<td>--</td>
<td>--</td>
<td>BusRd 6+64</td>
<td>BusRd 6+64</td>
<td>BusRdX 6+64</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>BusUpgr 6</td>
</tr>
<tr>
<td>M</td>
<td>BusWB 6 + 64</td>
<td>BusWB 6+64</td>
<td>NA</td>
<td>BusWB 6 + 64</td>
<td>--</td>
</tr>
</tbody>
</table>
Bandwidth of MSI vs. MESI

- 200 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?
- Difference between protocols is Exclusive State
  - Add BusUpgr for E->M transition
- Result is very small benefit!
  - Small number of E->M transitions
  - Only 6 bytes on bus

Bandwidth of MSI vs. MESI (1MB Cache)
**MSI BusUpgrd vs. BusRdX**

- **MSI S→M Transition Issues BusUpgrd**
  - could have block invalidated while waiting for BusUpgrd response
  - adds complexity to detect this
- **Instead just issue BusRdX**
  - from MESI put BusRdX in E→M and S→M
- **Result is 10% to 20% Improvement**
  - application dependent

**Cache Block Size**

- **Block size is unit of transfer and of coherence**
  - Doesn’t have to be, could have coherence smaller [Goodman]
- **Uniprocessor 3C’s**
  - (Compulsory, Capacity, Conflict)
- **SM adds Coherence Miss Type**
  - 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)
Cache Block Size: Traffic

Invalidates vs. Update

- **Pattern 1:**
  
  for i = 1 to k
  
  P1(write, x); // one write before reads
  
  P2--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
  
  end for j
  
  P2(read, x);
  
  end for i
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 = 1246 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 = 1400 bytes
  - Invalidate = 824 bytes

Invalidate vs. Update, cont.

- **What about real workloads?**
  - Update can generate too much traffic
  - Must limit (e.g., competitive snooping)

- **Current Assessment**
  - Update very hard to implement correctly
    (c.f., consistency discussion coming next)
  - 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?
Invalidate vs. Update: Miss Rate

Qualitative Sharing Patterns

- [Weber & Gupta, ASPLOS3]
- Read-Only
- Migratory Objects
  - Manipulated by one processor at a time
  - Often protected by a lock
  - Usually a write causes only a single invalidation
- Synchronization Objects
  - Often more processors imply more invalidations
- Mostly Read
  - More processors imply more invalidations, but writes are rare
- Frequently Read/Written
  - More processors imply more invalidations