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

Symmetric Multiprocessors Part 1

Copyright 2005 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),
J. P. Singh (Princeton), and Jim Smith (Wisconsin).
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

(d) Distributed-memory
Why Shared Memory?

• Pluses
  – For 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 relevent 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)

(Time)

- Process P1
- Process P2
- Load r2, x
- Interconnection Network
- Main Memory
Cache Coherence Problem (Step 2)

P1

P2

Id r2, x

Interconnection Network

Main Memory

ld r2, x

time

ld r2, x

Cache Coherence Problem (Step 3)

P1

P2

Id r2, x

Interconnection Network

Main Memory

ld r2, x

add r1, r2, r4

st x, r1
Snooping 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

Snooping 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**

![State Diagram](image)

(C) 2005 Mark D. Hill from Adve, Falsafi, Lebeck, Reinhardt, Singh, & Smith

CS/ECE 757 16
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
• What does state diagram look like?
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

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)
Invalidate 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?

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
Coherence vs. Consistency

- Intuition says loads should return latest value
  - what is latest?
- Coherence concerns only one memory location
- Consistency concerns apparent ordering for all locations

- A Memory System is Coherent if
  - can serialize all operations to that location such that,
  - operations performed by any processor appear in program order
    » program order = order defined by program text or assembly code
  - value returned a read is value written by last store to that location

Why Coherence != Consistency

/* initial A = B = flag = 0 */

P1
A = 1;
P2
while (flag == 0); /* spin */
B = 1;
flag = 1;
print A;
print B;

Intuition says printed A = B = 1
Coherence doesn’t say anything, why?
Consider coalescing write buffer
Sequential Consistency

- Lamport 1979

“A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program”

The Memory Model

Sequential processors issue memory ops in program order

P1 P2 Pn

Memory

switch randomly set after each memory op
Definitions and Sufficient Conditions

- **Sequentially Consistent Execution**
  - result is same as one of the possible interleavings on uniprocessor

- **Sequentially Consistent System**
  - any possible execution corresponds to some possible total order

Definitions

- **Memory operation**
  - execution of load, store, atomic read-modify-write access to mem location

- **Issue**
  - operation is issued when it leaves processor and is presented to memory system (cache, write-buffer, local and remote memories)

- **Perform**
  - store is performed wrt to a processor p when a load by p returns value produced by that store or a later store
  - A load is performed wrt to a processor when subsequent stores cannot affect value returned by that load

- **Complete**
  - memory operation is performed wrt all processors.

- **Program Execution**
  - Memory operations for specific run only (ignore non memory-referencing instructions)
Sufficient Conditions for Sequential Consistency

- Every processor issues memory ops in program order
- Processor must wait for store to complete before issuing next memory operation
- After load, issuing proc waits for load to complete, and store that produced value to complete before issuing next op
- Easily implemented with shared bus.

Synchronization

- Mutual Exclusion (critical sections)
  - Lock & Unlock
- Event Notification
  - point-to-point (producer-consumer, flags)
  - global (barrier)
- LOCK, BARRIER
  - How are these implemented?
Anatomy of A Synchronization Operation

- **Acquire Method**
  - method for trying to obtain the lock, or proceed past barrier

- **Waiting Algorithm**
  - Spin or busy wait
  - Block (suspend)

- **Release Method**
  - method to allow other processes to proceed past synchronization event

HW/SW Implementation Tradeoffs

- **User wants high level (ease of programming)**
  - \texttt{LOCK(lock\_variable)}, \texttt{UNLOCK(lock\_variable)}
  - \texttt{BARRIER(barrier\_variable, Num\_Procs)}

- **Hardware**
  - The Need for Speed (it’s fast)

- **Software**
  - Flexible

- **Want**
  - low latency
  - low traffic
  - Scalability
  - low storage overhead
  - fairness
How Not To Implement Locks

- **LOCK**
  ```c
  while(lock_variable == 1);
  lock_variable = 1;
  ```

- **UNLOCK**
  ```c
  lock_variable = 0;
  ```

- **Implementation requires Mutual Exclusion!**
  - Can have two processes successfully acquire the lock

---

Atomic Read-Modify-Write Operations

- **Test&Set(r,x)**
  ```c
  r = m[x]
  m[x] = 1
  ```

- **Swap(r,x)**
  ```c
  r = m[x], m[x] = r
  ```

- **Compare&Swap(r1,r2,x)**
  ```c
  if (r1 == m[x]) then
    r2 = m[x], m[x] = r2
  ```

- **Fetch&Op(r,x,op)**
  ```c
  r = m[x], m[x] = op(m[x])
  ```

(C) 2005 Mark D. Hill from Adve, Falsafi, Lebeck, Reinhardt, Singh, & Smith CS/ECE 757
Load-Locked Store-Conditional

- Pair of Instructions
- Load-Locked sets flag and address
- Store-Conditional fails if flag clear
- Flag is cleared on
  - invalidation
  - replacement
  - context switch

`lock: ll r1, location
    sc location, r2
    beqz r2, lock
    ret`

`unlock: st location, #0
    ret`

_____  

Alpha test & set Bit in 32-bit word

`t_and_set: # arg reg a0 holds bit# to test
  1:    ldl_l t0, (a1)  # get word to test
        bis zero, 1, t1  # logical OR set t1=1
        sll t1, a0, t2  # create bitmask
        and t0, t2, t3  # isolate bit
        bne t3, 2f  # branch if already set
        bis t0, t2, v0  # or in bit to be set
        sll t0, a0, v0  # conditional store
        beq v0, 3f  # branch if store failed
        mb  # memory barrier
        ret zero, (ra)  # ret 1 (v0) if set success
  2:    bis zero, zero, v0  # ret 0 if bit already set
        ret zero, (ra)
  3:    br zero, 1b  # retry interlocked update`
Performance of Test & Set

**LOCK**

\[
\text{while (test\&set(x) == 1);}
\]

**UNLOCK**

\[
x = 0;
\]

- High contention (many processes want lock)
- Remember the CACHE!
- Each test\&set is a read miss and a write miss
  - Not fair
- Problem is?
- Waiting Algorithm!

Better Lock Implementations

- Two choices:
  - Don't execute test\&set so much
  - Spin without generating bus traffic
- Test\&Set with Backoff
  - Insert delay between test\&set operations (not too long)
  - Exponential seems good (k^c_i)
  - Not fair
- Test-and-Test\&Set
  - Spin (test) on local cached copy until it gets invalidated, then issue test\&set
  - Intuition: No point in trying to set the location until we know that it's not set, which we can detect when it get invalidated...
  - Still contention after invalidate
  - Still not fair
**Fetch&Inc-Based Locks**

- **Ticket Lock**
  - **LOCK**
    - Obtain number via fetch&inc
    - Spin on now-serving counter
  - **Unlock**
    - Increment now-serving counter

- **Array based Lock**
  - Obtain location to spin on rather than value
  - Fair
  - Slight increase in storage
  - Put locations in separate cache blocks, else same traffic as t&t&s

---

**Queue-Based Locks**

- **Linked list points to next in line**
  - QOLB in hardware
  - MCS in software

(c) 2005 Mark D. Hill from Adve, Falsafi, Lebeck, Reinhardt, Singh, & Smith
What are the Performance Issues?

- Low latency:
  - should be able to get a free lock quickly
- Scalability:
  - should perform well beyond a small number of procs (< 64)
- Low storage overhead
- Fairness
- Blocking/Non-blocking
- Are spin locks fair?

Performance of Locks

- Contested vs. Uncontested
- Test&set is good with no contention
- Array based (Queue) is best with high contention
- Reactive Synchronization by Lim & Agarwal
  - Choose lock implementation based on contention
Implementation Details

- To Cache or Not to Cache, that is the question.

Uncached
- Latency for one operation increases
  + Fast hand-off between processes

Cached
- Might generate a lot of traffic if lock moves around
  + Might reuse lock a lot (locality), then traffic would be reduced by caching

- Must keep ownership for entire read-modify-write cycle
  - synchronization operation is visible to the memory system
  - we’ll exploit this fact later in the semester

Point-to-Point Event Synchronization

- Often use normal variables as flags
  
  ```
  a = f(x);     while (flag == 0);
  flag = 1;     b = g(a);
  ```

- If we know a before hand
  
  ```
  a = f(x)     while (a == 0);
  b = g(a);
  ```

- Assumes Sequential Consistency!!

- Full/Empty Bits
  - Set on Write
  - Cleared on Read
  - Can’t write if set, can’t read if clear
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 Locks, SW Combining Tree

Outline

• Motivation
• Coherence
• Coherence Tradeoffs
• Memory Consistency
• Synchronization