G22.2243-001
High Performance Computer Architecture

Lecture 11
Multiprocessing (Cont’d)

April 5, 2006
Outline

• Announcements
  – HW Assignment 4 due back today
  – Lab Assignment 4 due in a week: April 12
    \textbf{Deadline extended to April 19}
    No other extension will be possible

• Multiprocessors
  – Coherence protocols
    • Snooping-based protocols (review)
    • Directory-based protocols
  – Synchronization

[Hennessy/Patterson \textit{CA:QA} (3rd Edition): Chapter 6]
Snooping - Cache State Machine: Combined

State machine for CPU requests and bus requests for each cache block

- **Invalid**: Write miss for this block
- **Exclusive (read/write)**: CPU Read hit, CPU Write hit
- **Shared (read only)**: CPU Read miss
- **Write Miss**: Write back cache block, Place write miss on bus
- **Write back block**: Place read miss on bus
- **Read miss**: Write back block, Place read miss on bus
- **Write back block**: (abort memory access)
- **CPU Read**: Place read miss on bus
With A New State: Clean Exclusive (HW 4)
Larger Multiprocessors

- Separate Memory per Processor
- Local or Remote access via memory controller
- One Cache Coherency solution: non-cached pages
- Alternative: use a directory containing information for every block in memory
  - Which caches have a copy of block, dirty vs. clean, ...
- Info per memory block vs. per cache block?
  - Simpler protocol (centralized/one location)
  - Directory is $f$ (memory size x number of processors) vs. $f$ (cache size)
- Prevent directory as bottleneck?
  distribute directory entries with memory, each keeping track of which processors have copies of their memory blocks and in what state
Distributed Directory
Directory Protocol

• Similar to Snooping Protocol: Three states
  – **Shared**: ≥ 1 processor(s) have data, memory up-to-date
  – **Uncached**: (no processor has it; not valid in any cache)
  – **Exclusive**: 1 processor (owner) has data; memory out-of-date

• In addition to cache state, must track which processors have data when in the shared state (usually bit vector, 1 if processor has copy)

• Keep it simple(r):
  – Writes to non-exclusive data
    → write miss
  – Processor blocks until access completes
  – Assume messages received and acted upon in order sent
Directory Protocol (Cont’d)

• No bus and don’t want to broadcast:
  – interconnect no longer single arbitration point
  – all messages have explicit responses

• Typically 3 processors involved
  – Local node where a request originates
  – Home node where the memory location of an address resides
  – Remote node has a copy of a cache block, whether exclusive or shared

• Example messages on next slide:
P = processor number, A = address
# Directory Protocol Messages

<table>
<thead>
<tr>
<th>Message type</th>
<th>Source</th>
<th>Destination</th>
<th>Msg Content</th>
<th>Content Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Read miss</strong></td>
<td>Local cache</td>
<td>Home directory</td>
<td>P, A</td>
<td>Processor P has a read miss at address A; request data and make P a read sharer</td>
</tr>
<tr>
<td><strong>Write miss</strong></td>
<td>Local cache</td>
<td>Home directory</td>
<td>P, A</td>
<td>Processor P has a write miss at address A; request data and make P exclusive owner</td>
</tr>
<tr>
<td><strong>Invalidate</strong></td>
<td>Home directory</td>
<td>Remote caches</td>
<td>A</td>
<td>Invalidate a shared copy of data at address A</td>
</tr>
<tr>
<td><strong>Fetch</strong></td>
<td>Home directory</td>
<td>Remote cache</td>
<td>A</td>
<td>Fetch block at address A &amp; send it to its home directory; change state to shared at remote</td>
</tr>
<tr>
<td><strong>Fetch/Invalidate</strong></td>
<td>Home directory</td>
<td>Remote cache</td>
<td>A</td>
<td>Fetch block at address A &amp; send it to its home directory; invalidate the block in the cache</td>
</tr>
<tr>
<td><strong>Data value reply</strong></td>
<td>Home directory</td>
<td>Local cache</td>
<td>Data</td>
<td>Return a data value from the home memory</td>
</tr>
<tr>
<td><strong>Data write-back</strong></td>
<td>Remote cache</td>
<td>Home directory</td>
<td>A, Data</td>
<td>Write-back a data value for address A</td>
</tr>
</tbody>
</table>
State Transition Diagram for an Individual Cache Block in a Directory Based System

- States identical to snooping case
- Transactions very similar
- Transitions caused by read misses, write misses, invalidates, and data fetch requests
- Generates read miss & write miss messages to home directory
- Write misses that were broadcast on the bus for snooping
  - explicit invalidate & data fetch requests
CPU - Cache State Machine

- State machine for each Cache block

Invalid (Uncached)

- CPU Read hit
- Invalidate

Shared (read only)

- CPU Read
- Send Read Miss Message to h.d.

Exclusive (read/writ)

- CPU Write: Send Write Miss msg to h.d.
- CPU read miss: Send Read Miss

CPU read hit
- CPU write hit

CPU read miss:
- Send Data Write Back message and read miss to home directory

CPU write miss:
- Send Data Write Back message and Write Miss to home directory

Fetch/Invalidate
- send Data Write Back message to home directory

CPU Write:
- Send Write Miss message to home directory

Fetch:
- send Data Write Back message to home directory
State Transition Diagram for the Directory

• Same states & structure as the transition diagram for an individual cache
• Two actions: update of directory state & send messages to satisfy requests
• Keeps track of all copies of memory block
  – Uses a sharing set called Sharers
Example Directory Protocol

• Message sent to directory causes two actions:
  – Update the directory
  – More messages to satisfy request

• Block is in Uncached state: the copy in memory is the current value; only possible requests for that block are:
  – Read miss: requesting processor sent data from memory & requestor made (the first) sharing node; state of block made Shared
  – Write miss: requesting processor is sent the value. The block is made Exclusive to indicate that the only valid copy is cached. Sharers indicates the identity of the owner.

• Block is in Shared state: the memory value is up-to-date:
  – Read miss: requesting processor is sent back the data from memory & requesting processor is added to the sharing set.
  – Write miss: requesting processor is sent the value. All processors in the set Sharers are sent invalidate messages & Sharers is set to identity of requesting processor. The state of the block is made Exclusive.
Example Directory Protocol (Cont’d)

- Block is **Exclusive**: current value of the block is held in the cache of the processor identified by the set Sharers (the owner).

- Three possible directory requests:
  - **Read miss**: owner processor is sent a data fetch message, causing state of block in owner’s cache to transition to Shared and causes owner to send data to directory, where it is written to memory & sent back to requesting processor. Identity of requesting processor is added to set Sharers, which still contains the identity of the processor that was the owner (since it still has a readable copy); state is shared.

  - **Data write-back**: owner processor is replacing the block and hence must write it back, making memory copy up-to-date (the home directory essentially becomes the owner), the block is now Uncached, and the Sharer set is empty.

  - **Write miss**: block has a new owner. A message is sent to old owner (fetch/invalidate) causing the cache to send the value of the block to the directory from which it is sent to the requesting processor, which becomes the new owner. Sharers is set to identity of new owner, and state of block is made Exclusive. The old owner’s cache block status becomes Invalid.
Directory State Machine

- State machine for each memory block
- Uncached state if in memory

Diagram:

- Uncached
  - Data Write Back: Sharers = {} (Write back block)
  - Write Miss: Sharers = {P}; send Data Value Reply msg
  - Read miss: Sharers += {P}; send Data Value Reply

- Exclusive (read/writ)
  - Write Miss: Sharers = {P}; send Fetch/Invalidate; send Data Value Reply msg to remote cache
  - Write Miss: send Invalidate to Sharers; then Sharers = {P}; send Data Value Reply msg

- Shared (read only)
  - Write Miss: Sharers = {P}; send Fetch; send Data Value Reply msg
  - Read miss: Sharers += {P}; send Fetch; send Data Value Reply msg to remote cache (Write back block)

4/6/2006
Example

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Interconnect</th>
<th>Directory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>P1</strong></td>
<td><strong>P2</strong></td>
<td><strong>Bus</strong></td>
<td><strong>Directory</strong></td>
<td><strong>Memory</strong></td>
</tr>
<tr>
<td>P1: Write 10 to A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1: Read A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Read A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A1 and A2 map to the same cache block
**Example**

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Interconnect</th>
<th>Directory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>P1</strong></td>
<td><strong>P2</strong></td>
<td><strong>Bus</strong></td>
<td><strong>Directory</strong></td>
<td><strong>Memory</strong></td>
</tr>
<tr>
<td><strong>step</strong></td>
<td><strong>State</strong></td>
<td><strong>Addr</strong></td>
<td><strong>Value</strong></td>
<td><strong>State</strong></td>
</tr>
<tr>
<td>P1: Write 10 to A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>P1: Read A1</td>
<td></td>
<td></td>
<td>DaRp</td>
<td>P1</td>
</tr>
<tr>
<td>P2: Read A1</td>
<td></td>
<td></td>
<td></td>
<td>P1</td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td></td>
<td></td>
<td></td>
<td>P1</td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td></td>
<td></td>
<td></td>
<td>P1</td>
</tr>
</tbody>
</table>

A1 and A2 map to the same cache block
Example

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Interconnect</th>
<th>Directory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>step</strong></td>
<td><strong>State</strong></td>
<td><strong>Addr</strong></td>
<td><strong>Value</strong></td>
<td><strong>Action</strong></td>
</tr>
<tr>
<td>P1: Write 10 to A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td>WrMs</td>
</tr>
<tr>
<td>P1: Read A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>P2: Read A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A1 and A2 map to the same cache block
## Example

### Processor 1 Processor 2 Interconnect Directory Memory

<table>
<thead>
<tr>
<th>Step</th>
<th>Processor 1 Action</th>
<th>Processor 1 Addr</th>
<th>Processor 1 Value</th>
<th>Processor 2 Action</th>
<th>Processor 2 Addr</th>
<th>Processor 2 Value</th>
<th>Memory Addr</th>
<th>Memory Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Write 10 to A1</td>
<td>WrMs</td>
<td>P1</td>
<td>A1</td>
<td>A1</td>
<td>Ex</td>
<td>{P1}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td>DaRp</td>
<td>P1</td>
<td>A1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1: Read A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Read A1</td>
<td>RdMs</td>
<td>P2</td>
<td>A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shar.</td>
<td>A1</td>
<td>10</td>
<td>Ftch</td>
<td>P1</td>
<td>A1</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shar.</td>
<td>A1</td>
<td>10</td>
<td>DaRp</td>
<td>P2</td>
<td>A1</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1</td>
<td>Shar.</td>
<td>{P1,P2}</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Write Back**

A1 and A2 map to the same cache block
Example

A1 and A2 map to the same cache block
Example

A1 and A2 map to the same cache block

<table>
<thead>
<tr>
<th>Step</th>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Interconnect</th>
<th>Directory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>P1</strong></td>
<td><strong>P2</strong></td>
<td><strong>Bus</strong></td>
<td><strong>Directory</strong></td>
<td><strong>Memory</strong></td>
</tr>
<tr>
<td></td>
<td><strong>State</strong></td>
<td><strong>Addr</strong></td>
<td><strong>Value</strong></td>
<td><strong>Action</strong></td>
<td><strong>Proc.</strong></td>
</tr>
<tr>
<td>P1: Write 10 to A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td>WrMs</td>
<td>P1</td>
</tr>
<tr>
<td>P1: Read A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Read A1</td>
<td>Shar.</td>
<td>A1</td>
<td>10</td>
<td>RdMs</td>
<td>P2</td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td>Excl.</td>
<td>A1</td>
<td>20</td>
<td>Ftch</td>
<td>P1</td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td>Inv.</td>
<td>Inval.</td>
<td>P1</td>
<td>A1</td>
<td>Excl.</td>
</tr>
<tr>
<td>Excl. A2</td>
<td>40</td>
<td>DaRp</td>
<td>P2</td>
<td>A2</td>
<td>Excl.</td>
</tr>
</tbody>
</table>

P1: Write 20 to A1
WrMs P1 A1 A1 Ex {P1}  

P1: Read A1
Excl. A1 10 DaRp P1 A1 0

P2: Read A1
Shar. A1 10 Ftch P2 A1 10

P2: Write 20 to A1
Excl. A1 20 WrMs P2 A1 10

P2: Write 40 to A2
WrMs P2 A2 A2 Excl. {P2} 10
Implementing a Directory

• We assume operations atomic, but they are not; reality is much harder; must avoid deadlock when run out of buffers in network

• Optimization:
  – read miss or write miss in Exclusive: send data directly to requestor from owner vs. first to memory and then from memory to requestor
Synchronization

• Why Synchronize? Need to know when it is safe for different processes to use shared data

• Issues for Synchronization:
  – Uninterruptible instruction to fetch and update memory (atomic operation)
  – User level synchronization operation using this primitive
  – For large scale MPs, synchronization can be a bottleneck; techniques to reduce contention and latency of synchronization
Uninterruptable Instruction: Fetch and Update Memory

- **Atomic exchange**: interchange a value in a register for a value in memory
  - 0 => synchronization variable is free
  - 1 => synchronization variable is locked and unavailable
  - Set register to 1 & swap
  - New value in register determines success in getting lock
    - 0 if you succeeded in setting the lock (you were first)
    - 1 if other processor had already claimed access
  - Key is that exchange operation is indivisible
- **Test-and-set**: tests a value and sets it if the value passes the test
- **Fetch-and-increment**: it returns the value of a memory location and atomically increments it
  - 0 => synchronization variable is free
• Hard to have read & write in 1 instruction: use 2 instead
• **Load linked** (or load locked) + **store conditional**
  – Load linked returns the initial value
  – Store conditional returns 1 if it succeeds (no other store to same memory location since preceding load) and 0 otherwise

• **Example doing atomic swap with LL & SC:**
  
  ```
  try:  mov    R3,R4  ; move exchange value
        ll     R2,0(R1) ; load linked
        sc     R3,0(R1) ; store conditional
        beqz   R3,try  ; branch store fails (R3 = 0)
        mov    R4,R2   ; put load value in R4
  ```

• **Example doing fetch & increment with LL & SC:**
  
  ```
  try:  ll     R2,0(R1) ; load linked
        addi   R2,R2,#1 ; increment (OK if reg–reg)
        sc     R2,0(R1) ; store conditional
        beqz   R2,try  ; branch store fails (R2 = 0)
  ```
User Level Synchronization

- **Spin locks:** processor continuously tries to acquire, spinning around a loop trying to get the lock

  ```
  addi R2, R0, #1
  lockit:
  exch R2, 0(R1) ;atomic exchange
  bnez R2, lockit ;already locked?
  ```

- What about MP with cache coherency?
  - Want to spin on cache copy to avoid full memory latency
  - Likely to get cache hits for such variables

- Problem: exchange includes a write, which invalidates all other copies; this generates considerable bus traffic

- Solution: start by simply repeatedly reading the variable; when it changes, then try exchange (“test and test&set”):

  ```
  lockit:
  ld R2, 0(R1) ;load var
  bnez R2, lockit ;not free=>spin
  addi R2, R0, #1 ;load locked value
  exch R2, 0(R1) ;atomic exchange
  bnez R2, lockit ;already locked?
  ```
Memory Consistency Models

• Cache coherence ensures processors see a consistent view of memory

• What is consistency? How consistent the view should be?

• **When** must a processor see the new value? Consider the following:

  Assume both P1 and P2 have cached A and B with their initial value of zero

  P1: \( A = 0; \)  
  P2: \( B = 0; \)

  \( \ldots \)

  L1: if \((B == 0)\) ...

  \( \ldots \)

  L2: if \((A == 0)\) ...

• Impossible for both if statements L1 & L2 to be true?
  – What if write invalidate is delayed & processor continues?

• Memory consistency models:
  what are the rules for such cases?

• **Sequential consistency**: result of any execution is the same as if the accesses of each processor were kept in order and the accesses among different processors were interleaved \(\rightarrow\) assignments before ifs above
  – SC: delay all memory accesses until all invalidates done
Other Memory Consistency Models

• More relaxed models lead to faster execution
• Not really an issue for most programs as they are synchronized
  – A program is synchronized if all access to shared data are ordered by
    synchronization operations
    write (x)
    ...
    release (s) \{$unlock$\}
    ...
    acquire (s) \{$lock$\}
    ...
    read(x)
• Only those programs willing to be nondeterministic are not
  synchronized: outcome function of processor speed (data race)
• Several Relaxed Models for Memory Consistency since most programs
  are synchronized; characterized by their attitude towards: RAR, WAR,
  RAW, WAW to different addresses
Summary

• Caches contain all information on state of cached memory blocks
• Snooping and Directory Protocols similar
• Bus makes snooping easier because of broadcast
  – Uniform Memory Access
• Directory has extra data structure to keep track of state of all memory blocks
  – Distributing directory
  – Scalable shared address multiprocessor
  – Non Uniform Memory Access (NUMA)
Cross Cutting Issues: Performance Measurement of Parallel Processors

- Performance: how well scale as number of processors increases
- Speedup fixed as well as scaleup of problem
  - Assume benchmark of size \( n \) on \( p \) processors makes sense: how scale benchmark to run on \( m \times p \) processors?
  - **Memory-constrained scaling**: keeping the amount of memory used per processor constant
  - **Time-constrained scaling**: keeping total execution time, assuming perfect speedup, constant
- Example: 1 hour on 10 P, time \( \sim O(n^3) \), 100 P?
  - **Time-constrained scaling**: 1 hour, \( \Rightarrow 10^{1/3}n \Rightarrow 2.15n \) scale up
  - **Memory-constrained scaling**: 10n size \( \Rightarrow 10^3/10 \Rightarrow 100X \) or 100 hours!
    10X processors for 100X longer???
  - Need to know application well to scale: # iterations, error tolerance
Cross Cutting Issues:
Memory System Issues

- Multilevel cache hierarchy + multilevel inclusion -- every level of cache hierarchy is a subset of next level -- then can reduce contention between coherence traffic and processor traffic
  - Hard if cache blocks different sizes
- Also issues in memory consistency model and speculation, nonblocking caches, prefetching
Pitfall: Measuring MP performance by linear speedup v. execution time

- “linear speedup” graph of performance as scale CPUs
- Compare best algorithm on each computer
- **Relative speedup** - run same program on MP and uniprocessor
  - But parallel program may be slower on a uniprocessor than a sequential version
  - Or developing a parallel program will sometimes lead to algorithmic improvements, which should also benefit uni
- **True speedup** - run best program on each machine
- Can get **superlinear speedup** due to larger effective cache with more CPUs
Fallacy: Linear speedups are needed to make multiprocessors cost-effective

- Mark Hill & David Wood 1995 study
- Compare costs SGI uniprocessor and MP
- Uniprocessor = $38,400 + $100 * MB
- MP = $81,600 + $20,000 * P + $100 * MB
- 1 GB, uni = $138k v. mp = $181k + $20k * P
- What speedup for better MP cost performance?
  - 8 proc → $341k; $341k/138k ⇒ 2.5X
  - 16 proc → need only 3.6X, or 25% linear speedup
- Even if need some more memory for MP, not linear
Fallacy: Multiprocessors are “free”

• “Since microprocessors contain support for snooping caches, can build small-scale, bus-based multiprocessors for no additional cost”
• Need more complex memory controller (coherence) than for uniprocessor
• Memory access time always longer with more complex controller
• Additional software effort: compilers, operating systems, and debuggers all must be adapted for a parallel system