Last time in Lecture 17

Two kinds of synchronization between processors:

- **Producer-Consumer**
  - Consumer must wait until producer has produced value
  - Software version of a read-after-write hazard

- **Mutual Exclusion**
  - Only one processor can be in a critical section at a time
  - Critical section guards shared data that can be written

- **Producer-consumer synchronization implementable with just loads and stores, but need to know ISA’s memory model!**
- **Mutual-exclusion can also be implemented with loads and stores, but tricky and slow, so ISAs add atomic read-modify-write instructions to implement locks**
Recap: Sequential Consistency

A Memory Model

“A system 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 the order specified by the program”

Leslie Lamport

Sequential Consistency =

arbitrary order-preserving interleaving
of memory references of sequential programs

Sequential consistency imposes more memory ordering constraints than those imposed by uniprocessor program dependencies (→)

What are these in our example?

T1:
- Store (X), 1 (X = 1)
- Store (Y), 11 (Y = 11)

T2:
- Load R₁, (Y)
- Store (Y'), R₁ (Y' = Y)
- Load R₂, (X)
- Store (X'), R₂ (X' = X)

additional SC requirements
Relaxed Memory Model needs Fences

Producer posting Item x:
Load \( R_{\text{tail}} \) (tail)
Store \( (R_{\text{tail}}), x \)
\( \text{Membar}_{SS} \)
\( R_{\text{tail}} = R_{\text{tail}} + 1 \)
Store (tail), \( R_{\text{tail}} \)

Consumer:
Load \( R_{\text{head}} \) (head)
spin:
Load \( R_{\text{tail}} \) (tail)
if \( R_{\text{head}} = R_{\text{tail}} \) goto spin
\( \text{Membar}_{LL} \)
Load \( R_{\text{head}} \), \( (R_{\text{head}}) \)
\( R_{\text{head}} = R_{\text{head}} + 1 \)
Store (head), \( R_{\text{head}} \)

process(R)

4/11/2013 CS152, Spring 2013

Memory Coherence in SMPs

Suppose CPU-1 updates \( A \) to 200.
- **write-back**: memory and cache-2 have stale values
- **write-through**: cache-2 has a stale value

Do these stale values matter?
What is the view of shared memory for programming?
### Write-back Caches & SC

#### Program T1

<table>
<thead>
<tr>
<th>Memory</th>
<th>Cache-1</th>
<th>Cache-2</th>
<th>Prog T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X = 1</td>
<td>Y = 11</td>
<td>Y = 0</td>
<td>LD Y, R1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Y = 11</td>
<td>ST Y', R1</td>
</tr>
<tr>
<td>X' = Y'</td>
<td>X = 0</td>
<td>X = 0</td>
<td>LD X, R2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X = 11</td>
<td>ST X', R2</td>
</tr>
</tbody>
</table>

- T1 is executed
- Cache-1 writes back Y
- T2 executed
- Cache-1 writes back X
- Cache-2 writes back X' & Y'

### Write-through Caches & SC

#### Program T1

<table>
<thead>
<tr>
<th>Memory</th>
<th>Cache-1</th>
<th>Cache-2</th>
<th>Prog T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X = 0</td>
<td>Y = 10</td>
<td>Y = 0</td>
<td>LD Y, R1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Y = 10</td>
<td>ST Y', R1</td>
</tr>
<tr>
<td>X' = Y'</td>
<td>X = 0</td>
<td>X = 0</td>
<td>LD X, R2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X = 11</td>
<td>ST X', R2</td>
</tr>
</tbody>
</table>

- T1 executed
- T2 executed

**Write-through Caches don’t preserve sequential consistency either**
Maintaining Cache Coherence

- Hardware support is required such that
  - only one processor at a time has write permission for a location
  - no processor can load a stale copy of the location after a write
  $\Rightarrow$ cache coherence protocols

Cache Coherence vs. Memory Consistency

- A cache coherence protocol ensures that all writes by one processor are eventually visible to other processors, for one memory address
  - i.e., updates are not lost
- A memory consistency model gives the rules on when a write by one processor can be observed by a read on another, across different addresses
  - Equivalently, what values can be seen by a load
- A cache coherence protocol is not enough to ensure sequential consistency
  - But if sequentially consistent, then caches must be coherent
- Combination of cache coherence protocol plus processor memory reorder buffer used to implement a given architecture’s memory consistency model
Warmup: Parallel I/O

Either Cache or DMA can be the Bus Master and effect transfers

(DMA stands for “Direct Memory Access”, means the I/O device can read/write memory autonomous from the CPU)

Problems with Parallel I/O

Memory → Disk: Physical memory may be stale if cache copy is dirty

Disk → Memory: Cache may hold stale data and not see memory writes
### Snoopy Cache, *Goodman 1983*

- Idea: Have cache watch (or snoop upon) DMA transfers, and then “do the right thing”
- Snoopy cache tags are dual-ported

![Diagram of Snoopy Cache](image)

#### Snoopy Cache Actions for DMA

<table>
<thead>
<tr>
<th>Observed Bus Cycle</th>
<th>Cache State</th>
<th>Cache Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA Read Memory → Disk</td>
<td>Address not cached</td>
<td>No action</td>
</tr>
<tr>
<td></td>
<td>Cached, unmodified</td>
<td>No action</td>
</tr>
<tr>
<td></td>
<td>Cached, modified</td>
<td>Cache intervenes</td>
</tr>
<tr>
<td>DMA Write Disk → Memory</td>
<td>Address not cached</td>
<td>No action</td>
</tr>
<tr>
<td></td>
<td>Cached, unmodified</td>
<td>Cache purges its copy</td>
</tr>
<tr>
<td></td>
<td>Cached, modified</td>
<td>???</td>
</tr>
</tbody>
</table>

4/11/2013 CS152, Spring 2013
CS152 Administrivia

Shared Memory Multiprocessor

Use snoopy mechanism to keep all processors' view of memory coherent
Snoopy Cache Coherence Protocols

**write miss:**
the address is *invalidated* in all other caches *before* the write is performed

**read miss:**
if a dirty copy is found in some cache, a write-back is performed before the memory is read

---

Cache State Transition Diagram

*The MSI protocol*

Each cache line has state bits

<table>
<thead>
<tr>
<th>State bits</th>
<th>Address tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>M: Modified</td>
<td>S: Shared</td>
</tr>
<tr>
<td>I: Invalid</td>
<td></td>
</tr>
</tbody>
</table>

Write miss (P1 gets line from memory)
Other processor reads (P1 writes back)
Read miss (P1 gets line from memory)
Read by any processor

P₁ reads or writes
Other processor intent to write (P₁ writes back)
Other processor intent to write
Cache state in processor P₁
Two Processor Example
(Reading and writing the same cache line)

Observation

- If a line is in the M state then no other cache can have a copy of the line!
- Memory stays coherent, multiple differing copies cannot exist
**MESI: An Enhanced MSI protocol**

*increased performance for private data*

*Each cache line has a tag*

- **M**: Modified Exclusive
- **E**: Exclusive but unmodified
- **S**: Shared
- **I**: Invalid

<table>
<thead>
<tr>
<th>State bits</th>
<th>Address tag</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Write miss**
  - P₁ write or read
  - Other processor reads
  - P₁ writes back
- **Read miss, shared**
  - Read by any processor
- **Read by any processor**

- **P₁ intent to write**
  - Other processor reads
  - Other processor intent to write, P₁ writes back
  - Other processor intent to write
- **Cache state in processor P₁**

---

**Optimized Snoop with Level-2 Caches**

- Processors often have two-level caches
  - small L₁, large L₂ (usually both on chip now)
- **Inclusion property**: entries in L₁ must be in L₂
  - invalidation in L₂ $\Rightarrow$ invalidation in L₁
- Snooping on L₂ does not affect CPU-L₁ bandwidth

*What problem could occur?*
**Intervention**

When a read-miss for A occurs in cache-2, a read request for A is placed on the bus

- Cache-1 needs to supply & change its state to shared
- The memory may respond to the request also!

*Does memory know it has stale data?*

Cache-1 needs to intervene through memory controller to supply correct data to cache-2

**False Sharing**

A cache line contains more than one word

Cache-coherence is done at the line-level and not word-level

Suppose \( M_1 \) writes word\(_i\), and \( M_2 \) writes word\(_k\), and both words have the same line address.

*What can happen?*
Synchronization and Caches: Performance Issues

Cache-coherence protocols will cause mutex to ping-pong between P1’s and P2’s caches.

Ping-ponging can be reduced by first reading the mutex location (non-atomically) and executing a swap only if it is found to be zero.

Load-reserve & Store-conditional

Special register(s) to hold reservation flag and address, and the outcome of store-conditional

If the snooper sees a store transaction to the address in the reserve register, the reserve bit is set to 0

- Several processors may reserve ‘a’ simultaneously
- These instructions are like ordinary loads and stores with respect to the bus traffic

Can implement reservation by using cache hit/miss, no additional hardware required (problems?)
Out-of-Order Loads/Stores & CC

Blocking caches
One request at a time + CC $\Rightarrow$ SC

Non-blocking caches
Multiple requests (different addresses) concurrently + CC $\Rightarrow$ Relaxed memory models

CC ensures that all processors observe the same order of loads and stores to an address

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS252