CS 152 Computer Architecture and Engineering

Lecture 20: Snoopy Caches

Krste Asanovic
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www.eecs.berkeley.edu/~krste
http://inst.cs.berkeley.edu/~cs152
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
Recap: Sequential Consistency

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
Recap: Mutual Exclusion and Locks

Want to guarantee only one process is active in a critical section

- Blocking atomic read-modify-write instructions
  e.g., Test&Set, Fetch&Add, Swap
  vs
- Non-blocking atomic read-modify-write instructions
  e.g., Compare&Swap, Load-reserve/Store-conditional
  vs
- Protocols based on ordinary Loads and Stores
Issues in Implementing Sequential Consistency

Implementation of SC is complicated by two issues

- Out-of-order execution capability
  - \texttt{Load(a); Load(b)} \quad yes
  - \texttt{Load(a); Store(b)} \quad yes \text{ if } a \neq b
  - \texttt{Store(a); Load(b)} \quad yes \text{ if } a \neq b
  - \texttt{Store(a); Store(b)} \quad yes \text{ if } a \neq b

- Caches
  Caches can prevent the effect of a store from being seen by other processors

SC complications motivate architects to consider \textit{weak} or \textit{relaxed} memory models
Memory Fences
*Instructions to sequentialize memory accesses*

Processors with *relaxed or weak memory models* (i.e., permit Loads and Stores to different addresses to be reordered) need to provide *memory fence* instructions to force the serialization of memory accesses.

*Examples of processors with relaxed memory models:*
- Sparc V8 (TSO,PSO): Membar
- Sparc V9 (RMO):
  - Membar #LoadLoad, Membar #LoadStore
  - Membar #StoreLoad, Membar #StoreStore
- PowerPC (WO): Sync, EIEIO

*Memory fences are expensive operations, however, one pays the cost of serialization only when it is required*
Using Memory Fences

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

ensures that tail ptr is not updated before x has been stored

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

ensures that R is not loaded before x has been stored
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

- **T1 is executed**
  
<table>
<thead>
<tr>
<th>prog T1</th>
<th>cache-1</th>
<th>memory</th>
<th>cache-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST X, 1</td>
<td>X = 1</td>
<td>X = 0</td>
<td>Y = 11</td>
</tr>
<tr>
<td>ST Y, 11</td>
<td>Y = 11</td>
<td>Y = 10</td>
<td>X' = Y'</td>
</tr>
</tbody>
</table>

- **cache-1 writes back Y**

- **T2 executed**

- **cache-1 writes back X**

- **cache-2 writes back X' & Y'**

April 15, 2010

CS152, Spring 2010
Write-through Caches & SC

• T1 executed

• T2 executed

Write-through caches don’t preserve sequential consistency either
Cache Coherence vs. Memory Consistency

• A cache coherence protocol ensures that all writes by one processor are eventually visible to other processors
  – 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
  – 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 implements a given machine’s memory consistency model
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

⇒ cache coherence protocols
Warmup: Parallel I/O

(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](attachment:diagram.png)
### 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>
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>Address tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>state bits</td>
</tr>
</tbody>
</table>

M: Modified
S: Shared
I: Invalid

Write miss
(P1 gets line from memory)

Other processor reads
(P1 writes back)

Read miss
(P1 gets line from memory)

Read by any processor

Other processor writes

P1 reads or writes

P1 intent to write

Other processor intent to write
(P1 writes back)

Cache state in processor P1
Two Processor Example
(Reading and writing the same cache line)

P₁ reads
P₁ writes
P₂ reads
P₂ writes
P₁ reads
P₁ writes
P₂ writes
P₂ writes
P₁ writes

P₁ reads
P₁ writes
P₂ reads
P₂ writes
P₁ reads
P₁ writes
P₂ reads
P₂ writes
P₁ writes

M
S
I

P₂ reads, P₁ writes back
P₁ reads or writes
Write miss
P₁ intent to write
P₂ intent to write
P₂ reads, P₂ writes back
P₂ reads or writes
Write miss
P₁ intent to write
P₂ intent to write

Read miss
P₁ reads, P₂ writes back
P₂ intent to write
P₁ intent to write
P₂ reads
P₁ writes
P₂ reads
P₁ writes
P₂ reads
P₁ writes
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
Each cache line has a tag

<table>
<thead>
<tr>
<th>State Bits</th>
<th>Address Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>M: Modified Exclusive</td>
<td></td>
</tr>
<tr>
<td>E: Exclusive but unmodified</td>
<td></td>
</tr>
<tr>
<td>S: Shared</td>
<td></td>
</tr>
<tr>
<td>I: Invalid</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

P₁ write or read

Other processor reads
P₁ writes back

Other processor intent to write

Read miss, not shared

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

| state | blk addr | data0 | data1 | ... | dataN |

A cache block contains more than one word.

Cache-coherence is done at the block-level and not word-level.

Suppose \( M_1 \) writes \( \text{word}_i \) and \( M_2 \) writes \( \text{word}_k \) and both words have the same block address.

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

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

Ping-ponging can be reduced by first reading the \texttt{mutex} location (\textit{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

Load-reserve R, (a):
  <flag, adr> ← <1, a>;
  R ← M[a];

Store-conditional (a), R:
  if <flag, adr> == <1, a> then cancel other procs’ reservation on a;
  M[a] ← <R>;
  status ← succeed;
  else status ← fail;

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