Sequential Consistency and Cache Coherence Protocols

Arvind
Computer Science and Artificial Intelligence Lab
M.I.T.

Based on the material prepared by Arvind and Krste Asanovic
Memory Consistency 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?

November 9, 2005
Write-back Caches & SC

- **T1 is executed**
  - **ST X, 1**
  - **ST Y, 11**
  - `X = 1`
  - `Y = 11`

- **cache-1 writes back Y**
  - `X = 1`
  - `Y = 11`

- **T2 executed**
  - **LD Y, R1**
  - **ST Y', R1**
  - **ST X, R2**
  - **ST X', R2**

- **cache-1 writes back X**
  - `X = 1`
  - `Y = 11`

- **cache-2 writes back `X'` & `Y'`**
  - `X' = 0`
  - `Y' = 11`

---

**November 9, 2005**
Write-through Caches & SC

- T1 executed

- T2 executed

Write-through caches don’t preserve sequential consistency either
Maintaining Sequential Consistency

SC is sufficient for correct producer-consumer and mutual exclusion code (e.g., Dekker)

Multiple copies of a location in various caches can cause SC to break down.

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
A System with Multiple Caches

- Modern systems often have hierarchical caches
- Each cache has exactly one parent but can have zero or more children
- Only a parent and its children can communicate directly
- *Inclusion property* is maintained between a parent and its children, i.e.,
  \[ a \in L_i \implies a \in L_{i+1} \]
Cache Coherence Protocols for SC

**write request:**
the address is *invalidated (updated)* in all other caches *before (after)* the write is performed

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

*We will focus on Invalidation protocols as opposed to Update protocols*
Warmup: Parallel I/O

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

DMA stands for Direct Memory Access.
Problems with Parallel I/O

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

Disk → Memory: Cache may have data corresponding to the memory
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

```
\begin{figure}
\centering
\begin{tikzpicture}
  \node[draw,rectangle,minimum width=5cm,minimum height=7cm] (cache) at (0,0) {
    \textbf{Cache}
  };
  \node[draw,rectangle,minimum width=2cm,minimum height=2cm] (tags) at (1,1.5) {
    \textbf{Tags and State}
  };
  \node[draw,rectangle,minimum width=2cm,minimum height=2cm] (data) at (1,-1.5) {
    \textbf{Data (lines)}
  };
  \node[draw,rectangle,minimum width=2.5cm,minimum height=2.5cm] (proc) at (-3,0) {
    \textbf{Proc.}
  };
  \node[draw,rectangle,minimum width=1.5cm,minimum height=1.5cm] (snoop) at (3,0) {
    \textbf{Snoopy read port attached to Memory Bus}
  };

  \draw[->,thick] (proc) -- (cache);
  \draw[->,thick] (cache) -- (tags);
  \draw[->,thick] (cache) -- (data);
  \draw[->,thick] (cache) -- (snoop);

  \draw[->,thick] (proc) -- node[above] {A} (cache);
  \draw[->,thick] (cache) -- node[above] {A} (snoop);
  \draw[->,thick] (proc) -- node[right] {R/W} (cache);
  \draw[->,thick] (cache) -- node[right] {R/W} (snoop);
  \draw[->,thick] (proc) -- node[above] {D} (data);
  \draw[->,thick] (snoop) -- node[right] {R/W} (data);

  \node at (0,3) {Used to drive Memory Bus when Cache is Bus Master};
\end{tikzpicture}
\end{figure}
```
## Snoopy Cache Actions

<table>
<thead>
<tr>
<th>Observed Bus Cycle</th>
<th>Cache State</th>
<th>Cache Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Cycle 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>Write Cycle 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>

November 9, 2005
Use snoopy mechanism to keep all processors’ view of memory coherent
Cache State Transition Diagram

The MSI protocol

*Each* cache line has a tag

- **M**: Modified
- **S**: Shared
- **I**: Invalid

---

**State Bits**

Address tag

---

**State Diagram**

- **S** (Shared)
  - Read miss
  - Read by any processor
  - Other processor reads
  - \( P_1 \) writes back

- **M** (Modified)
  - \( P_1 \) intents to write
  - \( P_1 \) reads or writes
  - Write miss
  - Other processor intents to write

- **I** (Invalid)
  - Other processor intents to write

---

Cache state in processor \( P_1 \)

November 9, 2005
2 Processor Example

P_1 reads
P_1 writes
P_2 reads
P_2 writes
P_1 reads
P_1 writes
P_2 writes
P_1 writes

P_2 reads,
P_1 writes back

P_1 reads or writes
Write miss
P_2 intent to write

Read miss
P_1 intent to write

P_1 intent to write

P_2 reads,
P_2 writes back

P_2 reads
P_1 writes
Write miss
P_1 intent to write

Read miss
P_2 intent to write

P_1 reads or writes

P_2 reads

November 9, 2005
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

Each cache line has a tag

<table>
<thead>
<tr>
<th>State</th>
<th>M: Modified Exclusive</th>
<th>E: Exclusive, unmodified</th>
<th>S: Shared</th>
<th>I: Invalid</th>
</tr>
</thead>
</table>

Address tag

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

Cache state in processor $P_1$

- $P_1$ write or read
- $P_1$ write
- $P_1$ read
- Write miss
- Other processor intent to write
- Other processor reads
- $P_1$ writes back
- Read miss, shared
- $P_1$ intent to write
- Read by any processor
- Other processor intent to write
- November 9, 2005
Five-minute break to stretch your legs
Cache Coherence State Encoding

Valid and dirty bits can be used to encode S, I, and (E, M) states

- V=0, D=x ⇒ Invalid
- V=1, D=0 ⇒ Shared (not dirty)
- V=1, D=1 ⇒ Exclusive (dirty)
2-Level Caches

- Processors often have two-level caches
  - Small L1 on chip, large L2 off chip
- Inclusion property: entries in L1 must be in L2
  invalidation in L2 ⇒ invalidation in L1
- Snooping on L2 does not affect CPU-L1 bandwidth

What problem could occur?
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 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 `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.

November 9, 2005
Performance Related to Bus occupancy

In general, a \textit{read-modify-write} instruction requires two memory (bus) operations without intervening memory operations by other processors.

In a multiprocessor setting, bus needs to be locked for the entire duration of the atomic read and write operation:

\[\Rightarrow\] expensive for simple buses
\[\Rightarrow\] very expensive for split-transaction buses

Modern processors use
\textit{load-reserve store-conditional}
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

November 9, 2005
Performance:
Load-reserve & Store-conditional

The total number of memory (bus) transactions is not necessarily reduced, but splitting an atomic instruction into load-reserve & store-conditional:

- *increases bus utilization* (and reduces processor stall time), especially in split-transaction buses

- *reduces cache ping-pong effect* because processors trying to acquire a semaphore do not have to perform a store each time
Out-of-Order Loads/Stores & CC

Blocking caches
One request at a time + CC ⇒ SC

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

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

November 9, 2005
next time

Designing a Cache Coherence Protocol
Thank you !
2 Processor Example

Block b

P₁

P₁ write or read

P₁ write

P₂ reads, P₁ writes back

P₂ intent to write

P₂ intent to write

P₁ intent to write

P₂ intent to write

P₁ intent to write

P₂ reads, P₂ writes back

P₂ write or read

P₂ write

P₁ reads, P₂ writes back

P₁ write or read

P₂ write

P₁ write

Write miss

P₂ reads, P₁ writes back

Read miss

Read miss

Write miss

November 9, 2005