Sequential Consistency and Cache Coherence Protocols

Joel Emer
Computer Science and Artificial Intelligence Lab
M.I.T.

http://www.csg.csail.mit.edu/6.823
Synchronization

The need for synchronization arises whenever there are parallel processes in a system (even in a uniprocessor system)

- *Forks and Joins*: A parallel process may want to wait until several events have occurred

- *Producer-Consumer*: A consumer process must wait until the producer process has produced data

- *Exclusive use of a resource*: Operating system has to ensure that only one process uses a resource at a given time
A Producer-Consumer Example

Producer posting Item x:
  Load $R_{tail}$, (tail)
  Store $(R_{tail}), x$
  $R_{tail} = R_{tail} + 1$
  Store tail, $R_{tail}$

Consumer:
  Load $R_{head}$, (head)
  spin:
    Load $R_{tail}$, (tail)
    if $R_{head} == R_{tail}$ goto spin
    Load $R$, (R)
    $R_{head} = R_{head} + 1$
    Store head, $R_{head}$

process(R)

The program is written assuming instructions are executed in order.

Problems?
A Producer-Consumer Example

continued

Producer posting Item x:

1. Load $R_{tail}$, (tail)
2. Store (R$_{tail}$), x
   $R_{tail} = R_{tail} + 1$
3. Store tail, $R_{tail}$

Consumer:

1. Load $R_{head}$, (head)
2. spin:
3. Load $R_{tail}$, (tail)
   if $R_{head} == R_{tail}$ goto spin
4. Load R, ($R_{head}$)
   $R_{head} = R_{head} + 1$
   Store head, $R_{head}$

process(R)

Can the tail pointer get updated before the item x is stored?

Programmer assumes that if 3 happens after 2, then 4 happens after 1.

Problem sequences are:
2, 3, 4, 1
4, 1, 2, 3
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
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>
<th>prog T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST X, 1</td>
<td>X = 1</td>
<td>X = 0</td>
<td>Y = 11</td>
<td>LD Y, R1</td>
</tr>
<tr>
<td>ST Y, 11</td>
<td>Y = 11</td>
<td>Y = 10</td>
<td>X' = 11</td>
<td>ST Y', R1</td>
</tr>
</tbody>
</table>

• cache-1 writes back Y

• T2 executed

• cache-1 writes back X

• cache-2 writes back X' & Y'
Write-through Caches & SC

- **T1 executed**

  prog T1
  
  - ST X, 1
  - ST Y, 11

  ```
  prog T1
  ST X, 1
  ST Y, 11
  ```

  ```
  cache-1
  X= 0
  Y=10
  ```

  ```
  memory
  X = 0
  Y =10
  ```

  ```
  cache-2
  Y =
  Y’=
  ```

  ```
  X’=
  Y’=
  ```

- **T2 executed**

  ```
  prog T2
  LD Y, R1
  ST Y’, R1
  LD X, R2
  ST X’, R2
  ```

  ```
  cache-1
  X= 1
  Y=11
  ```

  ```
  memory
  X = 1
  Y =11
  ```

  ```
  cache-2
  Y = 11
  Y’= 11
  ```

  ```
  X’=
  Y’=
  ```

  ```
  X = 0
  ```

  ```
  X’=
  Y’=
  ```

  ```
  X’=
  Y’=
  ```

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

**Motivation:** We can do without locks -- SC is sufficient for writing producer-consumer and mutual exclusion codes (e.g., Dekker)

**Problem:** SC requires all processors to see writes occur in the same order, but multiple copies of a location in various caches can cause this to be violated.

To meet the ordering requirement it is sufficient for hardware to ensure:
- 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*
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* in all other caches *before* the write is performed, or
the address is *updated* in all other caches *after* the write is performed

**read request:**
if a dirty copy is found in some cache, that is the value that must be used, e.g., by doing a write-back and reading the memory or forwarding that dirty value directly to the reader.

*We will focus on Invalidation protocols as opposed to Update protocols*
Shared Memory Multiprocessor

Watch (snoop on) bus to keep all processors’ view of memory coherent
Snoopy Cache *Goodman 1983*

- Idea: Have the cache watch (or snoop upon) data transfers, and then “do the right thing”. Thus, memory operations are atomic with respect to all the caches.

Note: Snoopy cache tags have increased demand – often they are dual-ported.
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 data
- The memory may respond to the request also!

*Does memory know it has stale data?*

No, Cache-1 needs to intervene through memory controller to supply correct data to cache-2
## Snoopy Cache Actions

<table>
<thead>
<tr>
<th>Observed Bus Cycle</th>
<th>Cache State</th>
<th>Cache Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remote Read</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>Remote Write</td>
<td>Address not cached</td>
<td>No action</td>
</tr>
<tr>
<td></td>
<td>Cached, unmodified</td>
<td>Cache Purges Copy</td>
</tr>
<tr>
<td></td>
<td>Cached, modified</td>
<td>????????</td>
</tr>
</tbody>
</table>
Cache State Transition Diagram

The MSI protocol

Each cache line has a tag

<table>
<thead>
<tr>
<th>State</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>Modified</td>
</tr>
<tr>
<td>S</td>
<td>Shared</td>
</tr>
<tr>
<td>I</td>
<td>Invalid</td>
</tr>
</tbody>
</table>

Address tag

state bits

Cache state in processor \( P_1 \)

P_1 reads or writes

Write miss

Other processor intends to write

Read by any processor

Read miss

\( P_1 \) writes back

Other processor reads

\( P_1 \) intends to write

Other processor intends to write
2 Processor Example

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

P₁

P₂

P₁ reads, P₁ writes back
P₂ reads, P₂ writes back
P₁ intends to write
P₂ intends to write
Write miss
Write miss

P₁ reads or writes
P₂ reads or 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
MESI: An Enhanced MSI protocol
increased performance for private data

Each cache line has a tag

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

Each cache line has a tag

- Address tag
- State bits

Cache state in processor $P_1$

- $P_1$ write or read
- Other processor reads
  - $P_1$ writes back
- Read miss, shared
- Read by any processor

- $P_1$ write
- Other processor intends to write
- Other processor intends to write

- $P_1$ read
- Write miss
- Read miss, not shared

April 30, 2014

http://www.csg.csail.mit.edu/6.823

Sanchez & Emer
2 Processor Example
A cache block contains more than one word and cache-coherence is done at the block-level and not word-level.

Suppose \( P_1 \) writes word \( i \) and \( P_2 \) writes word \( k \) and both words have the same block address.

What can happen? The block may be invalidated (ping pong) many times unnecessarily because the addresses are in same block.
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.
In general, an atomic \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:
- \(\implies\) expensive for simple buses
- \(\implies\) \textit{very expensive} for split-transaction buses

Modern processors use

- \textit{load-reserve}
- \textit{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
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 stores each time
2-Level On-chip Caches

- **Inclusion property**: entries in L1 must be in L2
  invalidation in L2 $\Rightarrow$ invalidation in L1

- Does snooping on L2 affect CPU-L1 bandwidth?
  - yes -- to check if a dirty copy is stored in L1
- How can this be avoided?
  - Write-through L1 cache
Implementing SC

1. The memory operations of each individual processor appear to all processors in the order the requests are made to the memory.
   - Provided by cache coherence, which ensures that all processors observe the same order of loads and stores to an address

2. Any execution is the same as if the operations of all the processors were executed in some sequential order
   - Provided by enforcing a dependence between each memory operation and the following one.
SC Data Dependence

• Stall
  – Use in-order execution with blocking cache
    • Cache coherence plus allowing a processor to have only one request in flight at a time will provide SC

• Change architecture ⇒ Relaxed memory models
  – Use OOO and non-blocking caches
    • Cache coherence and allowing multiple requests (different addresses) concurrently gives high performance, then add fence operations to force ordering when needed

• Speculate...
Sequential Consistency Speculation

- Local load-store ordering uses standard OOO mechanism

- Globally **non-speculative** stores
  - Stores execute at commit -> stores are in-order!

- Globally **speculative** loads
  - **Guess** at issue that the memory location used by a load will not change between issue and commit of the instruction
    - this is equivalent to loads happening in-order at commit
  - **Check** at commit by remembering all loads addresses starting at issue and watching for writes to that location.
  - **Data Management** for rollback relies on the basic out-of-order speculative data management used for uni-processor rollback and instruction re-execution.
SC Speculative Behavior

1: ST A
2: LD A
3: LD A
4: ST A

CPU A

CPU B
Next lecture:
How to design a cache coherence protocol