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 \( R_1 \), (Y)
- Store \( (Y') \), \( R_1 \) \( (Y' = Y) \)
- Load \( R_2 \), (X)
- Store \( (X') \), \( R_2 \) \( (X' = X) \)

→ 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**
  
<table>
<thead>
<tr>
<th>Operation</th>
<th>SC Violation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load(a); Load(b)</td>
<td>yes</td>
</tr>
<tr>
<td>Load(a); Store(b)</td>
<td>yes if a ≠ b</td>
</tr>
<tr>
<td>Store(a); Load(b)</td>
<td>yes if a ≠ b</td>
</tr>
<tr>
<td>Store(a); Store(b)</td>
<td>yes if a ≠ b</td>
</tr>
</tbody>
</table>

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

SC complications motivates architects to consider **weak or 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 $R_{tail}$, (tail)
- Store ($R_{tail}$), x
- Membar$_{SS}$ $R_{tail}=R_{tail}+1$
- Store (tail), $R_{tail}$

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

**Consumer:**
- Load $R_{head}$, (head)
- spin:
  - Load $R_{tail}$, (tail)
  - if $R_{head}==R_{tail}$ goto spin
  - Membar$_{LL}$
- Load $R$, ($R_{head}$)
- $R_{head}=R_{head}+1$
- Store (head), $R_{head}$
- process(R)

**ensures that R is not loaded before x has been stored**

### 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?**
Write-back Caches & SC

- **T1** is executed
  - prog T1
    - ST X, 1
    - ST Y, 11
  - cache-1
    - X = 1
    - Y = 11
  - memory
    - X = 0
    - Y = 10
    - X' = 
    - Y' = 
  - cache-2
    - Y = 
    - Y' = 
    - X = 
    - X' = 

- cache-1 writes back **Y**
  - Y = 11
  - Y' = 11

- **T2** executed
  - prog T2
    - LD Y, R1
    - ST Y', R1
    - LD X, R2
    - ST X', R2
  - cache-1
    - X = 1
    - Y = 11
  - memory
    - X = 0
    - Y = 11
    - X' = 
    - Y' = 
  - cache-2
    - Y = 11
    - Y' = 11
    - X = 0
    - X' = 0

- cache-1 writes back **X**
  - X = 1
  - Y = 11
  - X' = 0
  - Y' = 11

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

Write-through Caches & SC

- **T1** executed
  - prog T1
    - ST X, 1
    - ST Y, 11
  - cache-1
    - X = 0
    - Y = 10
    - X' = 
    - Y' = 
  - memory
    - X = 0
    - Y = 10
    - X' = 
    - Y' = 
  - cache-2
    - Y = 
    - Y' = 
    - X = 
    - X' = 

- **T2** executed
  - prog T2
    - LD Y, R1
    - ST Y', R1
    - LD X, R2
    - ST X', R2
  - cache-1
    - X = 1
    - Y = 11
    - X' = 
    - Y' = 
  - memory
    - X = 1
    - Y = 11
    - X' = 0
    - Y' = 11
  - cache-2
    - Y = 11
    - Y' = 11
    - X = 0
    - X' = 0

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*

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 hold state 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 Components](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></td>
</tr>
<tr>
<td></td>
<td>Cached, unmodified</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cached, modified</td>
<td></td>
</tr>
<tr>
<td>DMA Write Disk ➔ Memory</td>
<td>Address not cached</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cached, unmodified</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cached, modified</td>
<td></td>
</tr>
</tbody>
</table>
CS152 Administrivia

- Quiz 5, Thursday April 23
  - Covers VLIW, Vector, Multithreaded

Shared Memory Multiprocessor

Use snoopy mechanism to keep all processors' view of memory coherent
Cache State Transition Diagram

Each cache line has a tag

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

Address tag

State bits

- Other processor reads
- P\textsubscript{1} writes back
- Read miss
- Read by any processor

Write miss

P\textsubscript{1} reads or writes

Other processor intent to write

Cache state in processor P\textsubscript{1}

Two Processor Example
(Reading and writing the same cache line)

- P\textsubscript{1} reads
- P\textsubscript{1} writes
- P\textsubscript{2} reads
- P\textsubscript{2} writes
- P\textsubscript{1} reads
- P\textsubscript{1} writes
- P\textsubscript{2} writes

Read miss

P\textsubscript{2} reads, P\textsubscript{1} writes back

P\textsubscript{1} intent to write

P\textsubscript{2} reads

P\textsubscript{2} intent to write

Write miss

P\textsubscript{1} reads or writes

P\textsubscript{2} reads or writes

P\textsubscript{1} reads

P\textsubscript{2} reads

P\textsubscript{1} reads

P\textsubscript{2} writes

P\textsubscript{1} 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

Cache state in processor $P_1$
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 ⇒ 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 block contains more than one word

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

Suppose $M_1$ writes word$_i$ and $M_2$ writes 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.
Performance Related to Bus Occupancy

In general, a *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

- expensive for simple buses
- *very expensive* for split-transaction buses

Modern ISAs use:

- *load-reserve*
- *store-conditional*

Load-reserve & Store-conditional

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

\[
\text{Load-reserve R, (a):} \\
\quad \langle \text{flag, adr} \rangle \leftarrow \langle 1, a \rangle; \\
\quad R \leftarrow M[a];
\]

\[
\text{Store-conditional (a), R:} \\
\quad \text{if} \ \langle \text{flag, adr} \rangle == \langle 1, a \rangle \\
\quad \text{then} \ \cancel{\text{other procs’ reservation on a;}} \\
\quad \quad M[a] \leftarrow \langle R \rangle; \\
\quad \quad \text{status} \leftarrow \text{succeed}; \\
\quad \text{else} \ \text{status} \leftarrow \text{fail};
\]

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?)
**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
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