Lecture 7: Consistency Models

- Topics: sequential consistency, requirements to implement sequential consistency, relaxed consistency models
Coherence Vs. Consistency

• Recall that coherence guarantees (i) that a write will eventually be seen by other processors, and (ii) write serialization (all processors see writes to the same location in the same order)

• The consistency model defines the ordering of writes and reads to different memory locations – the hardware guarantees a certain consistency model and the programmer attempts to write correct programs with those assumptions
### Example Programs

Initially, $A = B = 0$

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>$A = 1$</td>
<td>$B = 1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if ($B == 0$)</td>
<td>if ($A == 0$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>critical section</td>
<td>critical section</td>
<td></td>
</tr>
</tbody>
</table>

Initially, $A = B = 0$

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>$A = 1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>if ($A == 1$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$B = 1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>if ($B == 1$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>register = $A$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$P1$:
- Data = 2000
- Head = 1

$P2$:
- while (Head == 0)
  - {}
  - ...
  - = Data
### Sequential Consistency

We assume:
- Within a program, program order is preserved
- Each instruction executes atomically
- Instructions from different threads can be interleaved arbitrarily

Valid executions:
- abAcBCDdeE… or ABCDEFabGc… or abcAdBe… or aAbBcCdDeE… or …..
Sequential Consistency

- Programmers assume SC; makes it much easier to reason about program behavior

- Hardware innovations can disrupt the SC model

- For example, if we assume write buffers, or out-of-order execution, or if we drop ACKS in the coherence protocol, the previous programs yield unexpected outputs
Consistency Example - 1

- Consider a multiprocessor with bus-based snooping cache coherence and a write buffer between CPU and cache

<table>
<thead>
<tr>
<th>Initially A = B = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
</tr>
<tr>
<td>A ← 1</td>
</tr>
<tr>
<td>P2</td>
</tr>
<tr>
<td>B ← 1</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>if (B == 0)</td>
</tr>
<tr>
<td>Crit. Section</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>if (A == 0)</td>
</tr>
<tr>
<td>Crit. Section</td>
</tr>
</tbody>
</table>

The programmer expected the above code to implement a lock – because of write buffering, both processors can enter the critical section

The consistency model lets the programmer know what assumptions they can make about the hardware’s reordering capabilities
Consistency Example - 2

Sequential consistency requires program order
-- the write to Data has to complete before the write to Head can begin
-- the read of Head has to complete before the read of Data can begin
Consistency Example - 3

Initially, $A = B = 0$

\begin{align*}
\text{P1} & & \text{P2} & & \text{P3} \\
A &= 1 & \text{if } (A == 1) & B &= 1 \\
& & \text{if } (B == 1) & & \text{register} = A
\end{align*}

Sequential consistency can be had if a process makes sure that everyone has seen an update before that value is read – else, write atomicity is violated.
A multiprocessor is sequentially consistent if the result of the execution is achieveable by maintaining program order within a processor and interleaving accesses by different processors in an arbitrary fashion.

The multiprocessors in the previous examples are not sequentially consistent.

Can implement sequential consistency by requiring the following: program order, write serialization, everyone has seen an update before a value is read – very intuitive for the programmer, but extremely slow.
Relaxed Consistency Models

• We want an intuitive programming model (such as sequential consistency) and we want high performance

• We care about data races and re-ordering constraints for some parts of the program and not for others – hence, we will relax some of the constraints for sequential consistency for most of the program, but enforce them for specific portions of the code

• Fence instructions are special instructions that require all previous memory accesses to complete before proceeding (sequential consistency)
Fences

P1
{
  Region of code with no races
}
Fence
Acquire_lock
Fence
{
  Racy code
}
Fence
Release_lock
Fence

P2
{
  Region of code with no races
}
Fence
Acquire_lock
Fence
{
  Racy code
}
Fence
Release_lock
Fence


Other Performance Optimizations

- Program order is a major constraint – the following try to get around this constraint without violating seq. consistency
  - if a write has been stalled, prefetch the block in exclusive state to reduce traffic when the write happens
  - allow out-of-order reads with the facility to rollback if the ROB detects a violation (detected by re-executing the read later)
Potential Relaxations

• Program Order: (all refer to different memory locations)
  ➢ Write to Read program order
  ➢ Write to Write program order
  ➢ Read to Read and Read to Write program orders

• Write Atomicity: (refers to same memory location)
  ➢ Read others’ write early

• Write Atomicity and Program Order:
  ➢ Read own write early
## Relaxations

<table>
<thead>
<tr>
<th>Relaxation</th>
<th>W → R Order</th>
<th>W → W Order</th>
<th>R →RW Order</th>
<th>Rd others’ Wr early</th>
<th>Rd own Wr early</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM 370</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSO</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>PC</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

- IBM 370: a read can complete before an earlier write to a different address, but a read cannot return the value of a write unless all processors have seen the write.
- SPARC V8 Total Store Ordering (TSO): a read can complete before an earlier write to a different address, but a read cannot return the value of a write by another processor unless all processors have seen the write (it returns the value of own write before others see it).
- Processor Consistency (PC): a read can complete before an earlier write (by any processor to any memory location) has been made visible to all.
Performance Comparison

- Taken from Gharachorloo, Gupta, Hennessy, ASPLOS’91

- Studies three benchmark programs and three different architectures:
  - MP3D: 3-D particle simulator
  - LU: LU-decomposition for dense matrices
  - PTHOR: logic simulator

  - LFC: aggressive; lockup-free caches, write buffer with bypassing
  - RDBYP: only write buffer with bypassing
  - BASIC: no write buffer, no lockup-free caches
Performance Comparison

Figure 3: Relative performance of models on LFC

Figure 7: Performance of MP3D under LFC, RDBYP, and BASIC implementations.
Summary

• Sequential Consistency restricts performance (even more when memory and network latencies increase relative to processor speeds)

• Relaxed memory models relax different combinations of the five constraints for SC

• Most commercial systems are not sequentially consistent and rely on the programmer to insert appropriate fence instructions to provide the illusion of SC
Title

• Bullet