Lecture: Synchronization, Consistency Models

- Topics: synchronization wrap-up, need for sequential consistency, fences
Load-Linked and Store Conditional

- LL-SC is an implementation of atomic read-modify-write with very high flexibility

- LL: read a value and update a table indicating you have read this address, then perform any amount of computation

- SC: attempt to store a result into the same memory location, the store will succeed only if the table indicates that no other process attempted a store since the local LL (success only if the operation was “effectively” atomic)

- SC implementations do not generate bus traffic if the SC fails – hence, more efficient than test&test&set
Lock Vs. Optimistic Concurrency

lockit:    LL         R2, 0(R1)
           BNEZ    R2, lockit
           DADDUI R2, R0, #1
           SC         R2, 0(R1)
           BEQZ    R2, lockit
           Critical Section
           ST         0(R1), #0

tryagain: LL         R2, 0(R1)
           DADDUI R2, R2, R3
           SC         R2, 0(R1)
           BEQZ    R2, tryagain

LL-SC is being used to figure out if we were able to acquire the lock without anyone interfering – we then enter the critical section.

If the critical section only involves one memory location, the critical section can be captured within the LL-SC – instead of spinning on the lock acquire, you may now be spinning trying to atomically execute the CS.
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>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A = 1$</td>
<td>$B = 1$</td>
</tr>
<tr>
<td>if $(B == 0)$</td>
<td>if $(A == 0)$</td>
</tr>
<tr>
<td>critical section</td>
<td>critical section</td>
</tr>
</tbody>
</table>

Initially, $A = B = 0$

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

Initially, Head = Data = 0

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data = 2000</td>
<td></td>
</tr>
<tr>
<td>while (Head == 0)</td>
<td>{ }</td>
</tr>
<tr>
<td>Head = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>... = Data</td>
</tr>
</tbody>
</table>

...
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 .....
Problem 1

- What are possible outputs for the program below?

Assume x=y=0 at the start of the program

Thread 1
- x = 10
- y = x+y
- Print y

Thread 2
- y=20
- x = y+x
Problem 1

• What are possible outputs for the program below?

Assume x=y=0 at the start of the program

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>a</td>
</tr>
<tr>
<td>x = 10</td>
<td>y=20</td>
</tr>
<tr>
<td>B</td>
<td>b</td>
</tr>
<tr>
<td>y = x+y</td>
<td>x = y+x</td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>Print y</td>
<td></td>
</tr>
</tbody>
</table>

Possible scenarios: 5 choose 2 = 10

<table>
<thead>
<tr>
<th>ABCab</th>
<th>ABaCb</th>
<th>ABabC</th>
<th>AaBCb</th>
<th>AaBbC</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
<td>20</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>AabBC</td>
<td>aABCb</td>
<td>aABbC</td>
<td>aAbBC</td>
<td>abABC</td>
</tr>
<tr>
<td>50</td>
<td>30</td>
<td>30</td>
<td>50</td>
<td>30</td>
</tr>
</tbody>
</table>
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 - I

• An ooo core will see no dependence between instructions dealing with A and instructions dealing with B; those operations can therefore be re-ordered; this is fine for a single thread, but not for multiple threads

Initially $A = B = 0$

P1                        P2
A $\leftarrow 1$           B $\leftarrow 1$
...                        ...
if ($B == 0$)               if ($A == 0$)
Crit. Section             Crit. Section

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

Initially, $A = B = 0$

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

If a coherence invalidation didn’t require ACKs, we can’t confirm that everyone has seen the value of A.
Sequential Consistency

• A multiprocessor is sequentially consistent if the result of the execution is achievable by maintaining program order within a processor and interleaving accesses by different processors in an arbitrary fashion.

• 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.

• This is very slow... alternatives:
  - Add optimizations to the hardware (e.g., verify loads)
  - Offer a relaxed memory consistency model and fences
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

\[
P1\begin{cases}
\text{Region of code with no races}
\end{cases}
\]

\[
\text{Fence}
\begin{array}{l}
\text{Acquire\_lock}
\end{array}
\text{Fence}
\]

\[
\begin{cases}
\text{Racy code}
\end{cases}
\]

\[
\text{Fence}
\begin{array}{l}
\text{Release\_lock}
\end{array}
\text{Fence}
\]

\[
P2\begin{cases}
\text{Region of code with no races}
\end{cases}
\]

\[
\text{Fence}
\begin{array}{l}
\text{Acquire\_lock}
\end{array}
\text{Fence}
\]

\[
\begin{cases}
\text{Racy code}
\end{cases}
\]

\[
\text{Fence}
\begin{array}{l}
\text{Release\_lock}
\end{array}
\text{Fence}
\]