Principles of Concurrency

Lecture 7
Sequential Consistency
Consistency

How do we reason about the order in which concurrent operations are performed

‣ particularly relevant for shared-memory abstractions
‣ what view do threads have about memory? Memory model

Simplest model: strict consistency

‣ Maintain program order among operations from individual processors
‣ Enforce a global wall-clock time ordering on operations
‣ Every operation is sequentialized against this ordering
‣ But, what does “global time” even mean in a concurrent setting?

T1 T2 T3
W(x, 0) W(x, 1) Can L1 read 0, L2 read 1?
R(x) //L1 Can L1 and L2 both read 0?
R(x) //L2 Can L1 and L2 both read 1?
Sequential Consistency

Alternative model: sequential consistency

- Maintain program order among operations from individual processors
- Maintain a single interleaved execution order among operations from all processors that respects per-thread program order
  - Memory operations execute atomically
  - No global clock

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>W(x,1)</td>
<td></td>
</tr>
<tr>
<td>R(x)</td>
<td>//L1</td>
</tr>
<tr>
<td>R(x)</td>
<td>//L2</td>
</tr>
</tbody>
</table>

L1 can read 0 or 1
L2 can read 0 if L1 reads 0 or 1
L2 cannot read 0 if L1 reads 1

Key points:
- Program order of individual threads must be maintained
- Data coherence must be respected:
  - Any read must return most recent visible write
Architecturally

Processors must ensure its previous memory operation completes before starting the next one

- Memory must provide explicit acknowledgement previous write has completed
- Caches must invalidate or update all cached copies

Writes to the same location must be made visible in the same order to all processors

- serialized

Value of a write cannot be written until all invalidates or updates acknowledged
Architecturally

Executed code must contain enough synchronization to prevent incorrect reorderings

<table>
<thead>
<tr>
<th>Initial: $[x]=0 \land [y]=0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>proc 0</td>
</tr>
<tr>
<td>MOV $[x]←$1</td>
</tr>
<tr>
<td>MFENCE</td>
</tr>
<tr>
<td>MOV EAX←$[y]$</td>
</tr>
<tr>
<td><strong>Forbid</strong>: EAX=0 $\land$ EBX=0</td>
</tr>
</tbody>
</table>

Naive enforcement of SC is expensive (~40% overhead on x86 over well-studied benchmarks)

Initially, $[100]=0$

At the end, $[100]=2$

<table>
<thead>
<tr>
<th>proc:0</th>
<th>proc:1</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCK; INC $[100]$</td>
<td>LOCK; INC $[100]$</td>
</tr>
</tbody>
</table>
Is this transformation correct under SC?

Can transform P1 into P2, but not vise versa
Common Subexpression Elimination

- In general, not sound under SC

T1

W(x,1)
W(y,1)
if (R(y) == 2) {
  W(y,2)
  print (R(x))
}

T2

if (R(x) == 1) {
  W(x,2)
  W(y,2)
}

Only one interleaving that results in "print" being called:

W(x,1), W(y,1), R(x)1, W(x,2), W(y,2), R(y)2, R(x)2, Pr(2)

Optimization yields different results

W(x,1), W(y,1), R(x)1, W(x,2), W(y,2), R(y)2, Pr(1)
Another Example ...

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T1(M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(x)*2 //r1</td>
<td>W(x,1)</td>
<td>R(x)*2 //r1</td>
</tr>
<tr>
<td>R(y) //r2</td>
<td>W(y,1)</td>
<td>R(y) //r2</td>
</tr>
<tr>
<td>R(x)*2 //r3</td>
<td></td>
<td>r3 = r1</td>
</tr>
</tbody>
</table>

The transformed program (T1(M) || T2) can observe r2 == 1 and r3 == 0, which is not possible under (T1 || T2)

Need to ensure that the value of x has not changed since its last read

T1

L1: R(x)*2 //r1
    R(y) //r2
    r3 = r1
    if (x modified since L1)
    R(x)*2 //r3

The modification check can be implemented by tracking cache coherence and invalidation messages
Transformations

- Transformations involving only thread-local variables and compiler-generated temporaries are always SC preserving.

- Some simple transformations on shared variables are also safe:
  - Two consecutive loads of the same variable can be replaced by a single load: preserves SC since we only need to consider interleavings of the original program in which no other threads executes between these loads.

  Redundant load: \( R(x) //r1; R(x) //r2 \) \( ==> R(x) //r1; r2 = r1 \)
  
  Forwarded load: \( W(x,r1); R(x) //r2 \) \( ==> W(x,r1); r2 = r1 \)
  
  Dead store: \( W(x,r1); W(x,r2) \) \( ==> W(x,r2) \)
  
  Redundant store: \( R(x) //r1; W(x,r1) \) \( ==> R(x) //r1 \)

Non-SC preserving transformations are those that change the order of memory accesses performed by a thread in a way that becomes visible to other threads.
Constant-Folding and Copy Propagation

\[
\begin{align*}
\text{W}(x, 1) \\
\text{R}(z) \quad // r_1 \\
\text{W}(y, r_1) \\
\text{R}(x) \quad // r_2
\end{align*}
\]
\[
\Rightarrow
\begin{align*}
\text{W}(x, 1) \\
\text{R}(z) \quad // r_1 \\
\text{W}(y, z) \\
\text{r}_2 = 1
\end{align*}
\]

Is there a concurrent context in which transformed program exhibits a behavior not possible in the original?

Consider:

if (R(x) == 1) {
    W(x, 3);
    W(z, 2)
} else {
    W(z, 4)
}

Original:  
\[
\begin{align*}
\text{r}_1 = 0, \quad \text{r}_2 = 1 \\
\text{r}_1 = 4, \quad \text{r}_2 = 1 \\
\text{r}_1 = 0, \quad \text{r}_2 = 3 \\
\text{r}_1 = 2, \quad \text{r}_2 = 3
\end{align*}
\]

Transformed:  
\[
\begin{align*}
\text{r}_1 = 0, \quad \text{r}_2 = 1 \\
\text{r}_1 = 4, \quad \text{r}_2 = 1 \\
\text{r}_1 = 0, \quad \text{r}_2 = 1 \\
\text{r}_1 = 2, \quad \text{r}_2 = 1
\end{align*}
\]