Memory Consistency
A Crash Course

Brandon Lucia
CSE 471
Memory Consistency Model

Informal Definition:

“Defines the value a read operation may read at each point during the execution”
Memory Consistency Model

Informal Definition:

“Defines the value a read operation may read at each point during the execution”

“Defines the set of legal observable orders of memory operations during an execution”
Memory Consistency Model

Informal Definition:

“Defines the value a read operation may read at each point during the execution”

“Defines the set of legal observable orders of memory operations during an execution”

“Defines which reorderings of memory operations are permitted”
Review: Coherence

2 Invariants:

1) “One Writer or One or More Readers”

2) “Reading $X$ gets the value of the last write to $X$”
Review: Coherence

2 Invariants:

1) “One Writer or One or More Readers”

2) “Reading X gets the value of the last write to X”
Without Coherence
(The coherence invariants prevent this from happening)

Processors can’t decide who wrote last.
Green is hosed.
Coherence defines the set of legal orders of accesses to a single memory location.
Consistency is Ordering

Consistency defines the set of legal orders of accesses to multiple memory locations.
Sequential Consistency (SC)
The simplest, most intuitive memory consistency model

Two Invariants to SC:

Instructions are executed in program order

All processors agree on a total order of executed instructions
The SC “Switch”

Execution

Execute

Wr X
Rd Y

Wr Y
Rd X

Rd X
The SC “Switch”

Execution

Wr X

Wr Y

Rd X

Rd Y

Execute
The SC "Switch"

- **Execution**
  - Wr X
  - Rd Y

- **Wr X**
  - Rd Y

- **Wr Y**
  - Rd X

- **Rd X**
The SC “Switch”

```
Execute
Wr X
Rd Y
Wr Y
Rd X
Rd X
```
The SC “Switch”

**Execution**
- Wr X
- Rd Y
- Wr Y
- Rd X

**Execute**
The SC "Switch"

Execution

Wr X
Rd Y
Wr Y
Rd X
Rd X

Execute

Wr X
Rd Y
Wr Y
Rd X
Rd X
Why is SC Important?

Who cares?... **You** care!

SC is how *programmers* think.

<table>
<thead>
<tr>
<th><strong>Intuitive (SC)</strong></th>
<th><strong>Weird (not SC)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Wr X</strong></td>
<td><strong>Rd Y</strong></td>
</tr>
<tr>
<td><strong>Wr Y</strong></td>
<td><strong>Wr X</strong></td>
</tr>
<tr>
<td><strong>Rd X</strong></td>
<td><strong>Rd Y</strong></td>
</tr>
<tr>
<td><strong>Rd X</strong></td>
<td><strong>Rd X</strong></td>
</tr>
</tbody>
</table>

SC prohibits all reordering of instructions (Invariant 1)
Why are Instructions Reordered?
And when does it matter anyway?
Why are Instructions Reordered?

Optimization.
Reordering #1: Write Buffers

CPU can read its write buffer, but not others’

Buffered writes eventually end up in coherent shared memory
Reordering #1: Write Buffers

Program
Initially $X == Y == 0$

$X = 1$  $Y = 1$

$r1 = Y$  $r2 = X$

Is $r1 == r2 == 0$ a valid result?
Reordering #1: Write Buffers

Initially $X == Y == 0$

$X=1 \quad Y=1$

$r1=Y \quad r2=X$

Is $r1 == r2 == 0$ a valid result?

$r1 == r2 == 0$ is not SC, but it can happen with write buffers.
Reordering #1: Write Buffers

Program
Initially X == Y == 0

Execution

\[ r_1 = Y \quad r_2 = X \]
Reordering #1: Write Buffers

Program
Initially X == Y == 0

Execution

r1 = Y    r2 = X
Reordering #1: Write Buffers

Program
Initially X == Y == 0

Execution
r1 = Y
r2 = X
Reordering #1: Write Buffers

Program
Initially X == Y == 0

Execution

r2=X

r1=Y

M

M

X=1

Y=1
Reordering #1: Write Buffers

Program
Initially X == Y == 0

Execution
Reordering #1: Write Buffers

Program
Initially X == Y == 0

Execution
r1 = Y [r1 <= 0]
Reordering #1: Write Buffers

Program
Initially $X == Y == 0$

Execution
$r1 = Y$ [$r1 <- 0$
$r2 = X$ [$r2 <- 0$]
Reordering #1: Write Buffers

Program
Initially $X == Y == 0$

Execution
$\text{r1} = Y \ [r1 \ <- \ 0]$
$\text{r2} = X \ [r2 \ <- \ 0]$
$X = 1$
$Y = 1$
(Not SC!)

WBs let reads finish before older writes
Reordering #2: Write Combining

Coalescing Write Buffer

Program
X, Z in same $ line
X = 1
Y = 1
Z = 1

4 word cache line
Reordering #2: Write Combining

<table>
<thead>
<tr>
<th>Coalescing Write Buffer</th>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>X=1</td>
<td>X=1</td>
</tr>
<tr>
<td></td>
<td>Y=1</td>
</tr>
<tr>
<td></td>
<td>Z=1</td>
</tr>
</tbody>
</table>
Reordering #2: Write Combining

Coalescing Write Buffer

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X=1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Y=1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Program

X, Z in same $ line

X=1
Y=1
Z=1
Reordering #2: Write Combining

**Coalescing Write Buffer**

<table>
<thead>
<tr>
<th></th>
<th>X=1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Y=1</td>
<td></td>
<td>Z=1</td>
</tr>
</tbody>
</table>

**Program**

X,Z in same $ line

<table>
<thead>
<tr>
<th></th>
<th>X=1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Y=1</td>
<td></td>
<td>Z=1</td>
</tr>
</tbody>
</table>
Reordering #2: Write Combining

Combining the write to $X$ & $Z$ saves bandwidth, but **reorders** $Z=1$ and $Y=1$. 
Reordering #3: Compilers

The compiler hoists the write out of the loop, permitting new (non-SC) results (e.g., “1 0 0 0 0 0 0...”)

\[
\begin{align*}
X &= 0 \\
\text{for (i .. 100) } \\
X &= 1 \\
\text{print x}
\end{align*}
\]

\[
\begin{align*}
X &= 0 \\
\text{Compiler} \\
X &= 1 \\
\text{for (i .. 100) } \\
\text{print x}
\end{align*}
\]

\[
\begin{align*}
X &= 0 \\
\text{Been hoisted!}
\end{align*}
\]
When is Reordering a Problem?
When is Reordering a Problem?

When Executions Aren’t SC
When is an Execution Not SC?

When a memory operation happens before itself

**Execution**

\[ r_1 = Y [r_1 \gets 0] \]
\[ r_2 = X [r_2 \gets 0] \]
\[ X = 1 \]
\[ Y = 1 \]

**Happens-Before Graph**

\[ X = 1 \]
\[ Y = 1 \]
\[ r_1 = Y \]
\[ r_2 = X \]
When is an Execution Not SC?

When a memory operation happens before itself

Program Order HB Edge

Execution

\[
\begin{align*}
    r1 &= Y \ [r1 \leftarrow 0] \\
    r2 &= X \ [r2 \leftarrow 0] \\
    X &= 1 \\
    Y &= 1
\end{align*}
\]

Happens-Before Graph

\[
\begin{align*}
    X &= 1 \\
    Y &= 1 \\
    r1 &= Y \\
    r2 &= X
\end{align*}
\]
When is an Execution Not SC?

When a memory operation happens before itself

Execution

\[
\begin{align*}
  r1 &= Y \ [r1 \leftarrow 0] \\
  r2 &= X \ [r2 \leftarrow 0] \\
  X &= 1 \\
  Y &= 1 \\
\end{align*}
\]

Happens-Before Graph

\[
\begin{align*}
  X &= 1 \\
  Y &= 1 \\
  r1 &= Y \\
  r2 &= X \\
\end{align*}
\]

Program Order HB Edge

Causal Order HB Edge
When is an Execution Not SC?

When a memory operation happens before itself

Execution

r1=Y [r1 <- 0]

r2=X [r2 <- 0]

X=1

Y=1

Happens-Before Graph

X=1

Y=1

r1=Y

r2=X

If there is a cycle in the happens-before graph, the execution is not SC.
So... are Computers Wrong?!

SC is how **programmers** think.

SC prohibits **all** reordering of instructions

WBs let reads finish before older writes

Combining writes saves bandwidth but reorders writes
Relaxed Memory Consistency

Relaxed Memory Models permit reorderings, unlike SC
x86-TSO (intel x86s)

“The Write Buffer Memory Model”

Total Store Order - loads may complete before older stores to different locations complete.
PSO<sub>(SPARC)</sub>

“The Write Combining Memory Model”

Relaxes W→W order

Partial Store Order - loads and stores may complete before older stores to different locations complete.
In General

Starting with PSO and relaxing R→R and R→W yields Weak Ordering or Release Consistency (alpha)

Depending on the implementation
SC and Relaxed Consistency

SC is required for correctness and programmer sanity

+ Reordering is required* for performance

Goal: Ensure SC executions while permitting Relaxed Consistency reorderings

*Usually; the MIPS memory model is SC (surprising!)
How to ensure SC, but permit reordering?
Synchronization Prevents Reordering

Memory fences are another type of synchronization

Reordering prevented

Fence implementation depends on reordering implementation

TSO: Stall reads until write buffer is empty
Synchronization For Real Programmers

Memory fences are wrapped up in locks, etc.

Reordering prevented

Direct use of fences possible, but inadvisable.

USE A SYNCHRONIZATION LIBRARY
Data Races

Synchronization imposes happens-before on otherwise unordered operations

Data Race: Unordered operations to the same memory location, at least one a write
Memory Models across the System Stack

Language
Java/C++: SC for data-race-free programs

Compiler
Conservative with reordering when d-r-f can’t be proved

Architecture
Usually very weak for max optimization (lots of reordering)

Note: fences from “above” ensure SC