CS636: Memory Consistency Models

Swarnendu Biswas

Semester 2018-2019-II
CSE, IIT Kanpur

Content influenced by many excellent references, see References slide for acknowledgements.
Correctness of Shared-memory Programs

“To write correct and efficient shared memory programs, programmers need a precise notion of how memory behaves with respect to read and write operations from multiple processors”

Busy-Wait Paradigm

Object X = null;
boolean done = false;

Thread T1

X = new Object();
done = true;

Thread T2

while (!done) {}
if (X != null)
    X.compute();
What Value Can a Read Return?

Core C1

S1: store X, 10
S2: store done, 1

Core C2

L1: load r1, done
B1: if (r1 != 1) goto L1
L2: load r2, X

X = 0
done = 0
Reordering of Accesses by Hardware

- Store-store
- Load-load
- Load-store
- Store-load

Different addresses!
Reordering of Accesses by Hardware

Correct in a single-threaded context

Non-trivial in a multithreaded context

Store-load

Different addresses!
What values can a load return?

Return the “last” write

Uniprocessor: program order

Multiprocessor: ?
Memory Consistency Model

Set of rules that govern how systems process memory operation requests from multiple processors

- Determines the order in which memory operations appear to execute

Specify the allowed behaviors of multithreaded programs executing with shared memory

- Both at the hardware-level and at the programming-language-level
- There can be multiple correct behaviors
Importance of Memory Consistency Models

- Determines what optimizations are correct
- Contract between the programmer and the hardware
- Influences ease of programming and program performance
- Impacts program portability
Dekker’s Algorithm

Core C1
S1: store flag1, 1
L1: load r1, flag2

Core C2
S2: store flag2, 1
L2: load r2, flag1

Can both r1 and r2 be set to zero?
Issues with Memory Consistency

Visibility

• When does a value update become visible to others?

Ordering

• When can operations of any given thread appear out of order to another thread?
Sequential Consistency

A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all 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 (SC)

Uniprocessor

• operations executed in order specified by the program

Multiprocessor

• all operations executed in order, and the operations of each individual core appear in program order
Uniprocessor Memory Model

• Memory operations occur in program order
  • Only maintain data and control dependences
• Read from memory returns the value from the last write in program order
• Compiler optimizations preserve these semantics
Interleavings with SC

**TABLE 3.1:** Should r2 Always be Set to NEW?

<table>
<thead>
<tr>
<th>Core C1</th>
<th>Core C2</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1: Store data = NEW;</td>
<td>L1: Load r1 = flag;</td>
<td>/* Initially, data = 0 &amp; flag ≠ SET */</td>
</tr>
<tr>
<td>S2: Store flag = SET;</td>
<td>B1: if (r1 ≠ SET) goto L1;</td>
<td>/* L1 &amp; B1 may repeat many times */</td>
</tr>
<tr>
<td></td>
<td>L2: Load r2 = data;</td>
<td></td>
</tr>
</tbody>
</table>
Interleavings with SC

S1: data = NEW; /* NEW */

S2: flag = SET; /* SET */

L1: r1 = flag; /* 0 */

L1: r1 = flag; /* 0 */

L1: r1 = flag; /* 0 */

L1: r1 = flag; /* SET */

L2: r2 = data; /* NEW */
Every load gets its value from the last store before it (in global memory order) to the same address
Suppose we have two addresses \( a \) and \( b \):

- \( a == b \) or \( a != b \)

Constraints:

- If \( L(a) <_p L(b) \) \( \Rightarrow L(a) <_m L(b) \)
- If \( L(a) <_p S(b) \) \( \Rightarrow L(a) <_m S(b) \)
- If \( S(a) <_p S(b) \) \( \Rightarrow S(a) <_m S(b) \)
- If \( S(a) <_p L(b) \) \( \Rightarrow S(a) <_m L(b) \)
Challenges in Implementing SC

• Is preserving program order on a per-location basis sufficient?
Need for Write Atomicity

Core C1

\[ A = 1 \]

Core C2

\[
\begin{align*}
\text{if (A == 1)} \\
B &= 1
\end{align*}
\]

Core C3

\[
\begin{align*}
\text{if (B == 1)} \\
tmp &= A
\end{align*}
\]
Write Buffers with Bypassing

Core 1

S1: store flag1, 1
L1: load r1, flag2

Core 2

S2: store flag2, 1
L2: load r2, flag1

flag1 = 0
flag2 = 0
SC in Architecture with Caches

• Replication of data requires a cache coherence protocol
  • Several definitions of cache coherence protocols exist
• Propagating new values to multiple other caches is non-atomic
## Serialization of Writes

<table>
<thead>
<tr>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
<th>Core 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A = 1 )</td>
<td>( A = 2 )</td>
<td>while (( B != 1 )) {}</td>
<td>while (( B != 1 )) {}</td>
</tr>
<tr>
<td>( B = 1 )</td>
<td>( C = 1 )</td>
<td>while (( C != 1 )) {}</td>
<td>while (( C != 1 )) {}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{tmp1} = A )</td>
<td>( \text{tmp2} = A )</td>
</tr>
</tbody>
</table>
### Serialization of Writes

<table>
<thead>
<tr>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
<th>Core 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = 1</td>
<td>A = 2</td>
<td>while (B != 1) {}</td>
<td>while (B != 1) {}</td>
</tr>
<tr>
<td>B = 1</td>
<td>C = 1</td>
<td>while (C != 1) {}</td>
<td>while (C != 1) {}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp1 = A</td>
<td>tmp2 = A</td>
</tr>
</tbody>
</table>

Cache coherence must serialize writes to the same memory location

Writes to the same memory location must be seen in the same order by all
Cache Coherence

Single writer multiple readers (SWMR)

Memory updates are passed correctly, cached copies always contain the most recent data

Virtually a synonym for SC, but for a single memory location

Alternate definition based on relaxed ordering

- A write is **eventually** made visible to all processors
- Writes to the **same** location appear to be seen in the same order by all processors (serialization)
  - SC - *all*
Memory Consistency vs Cache Coherence

**Memory Consistency**

- **Defines** shared memory behavior
- Related to **all** shared-memory locations
- Policy on **when** new value is propagated to other cores
- Memory consistency implementations can use cache coherence as a “black box”

**Cache Coherence**

- **Does not define** shared memory behavior
- Specific to a **single** shared-memory location
- **Propagate** new value to other cached copies
  - Invalidation-based or update-based
End-to-end SC

Simple memory model that can be implemented both in hardware and in languages

Performance can take a hit

- Naive hardware
- Maintain program order - expensive for a write
SC-Preserving Optimizations

• Redundant load  
  \[ t = X; u = X; \quad \iff \quad t = X; u = t; \]

• Forwarded load  
  \[ X = t; u = X; \quad \iff \quad X = t; u = t; \]

• Dead store  
  \[ X = t; X = u; \quad \iff \quad X = u; \]

• Redundant store  
  \[ t = X; X = t; \quad \iff \quad t = X; \]
Optimizations Forbidden in SC

- Loop invariant code motion
- Common sub-expression elimination
- ...

<table>
<thead>
<tr>
<th>Original</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1: t = X*2</td>
<td>L1: t = X*2</td>
</tr>
<tr>
<td>L2: u = Y</td>
<td>L2: u = Y</td>
</tr>
<tr>
<td>L3: v = X*2</td>
<td>03: v = t</td>
</tr>
</tbody>
</table>
## Optimizations Forbidden in SC

- Loop invariant code motion
- Common sub-expression elimination
- ...

<table>
<thead>
<tr>
<th>Original</th>
<th>Optimized</th>
<th>Concurrent</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1: $t = X \times 2$</td>
<td>L1: $t = X \times 2$</td>
<td>C1: $X = 1$</td>
</tr>
<tr>
<td>L2: $u = Y$</td>
<td>L2: $u = Y$</td>
<td>C2: $Y = 1$</td>
</tr>
<tr>
<td>L3: $v = X \times 2$</td>
<td>O3: $v = t$</td>
<td></td>
</tr>
</tbody>
</table>

## Optimizations Forbidden in SC

<table>
<thead>
<tr>
<th>Original</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1: X = 1</td>
<td>L1: X = 1</td>
</tr>
<tr>
<td>L2: P = Q</td>
<td>L2: P = Q</td>
</tr>
<tr>
<td>L3: t = X</td>
<td>L3: t = 1</td>
</tr>
<tr>
<td>L1: X = 1</td>
<td>L1: ;</td>
</tr>
<tr>
<td>L2: P = Q</td>
<td>L2: P = Q</td>
</tr>
<tr>
<td>L3: X = 2</td>
<td>L3: X = 2</td>
</tr>
<tr>
<td>L1: t = X</td>
<td>L1: t = X</td>
</tr>
<tr>
<td>L2: P = Q</td>
<td>L2: P = Q</td>
</tr>
<tr>
<td>L3: X = t</td>
<td>L3: ;</td>
</tr>
</tbody>
</table>

- **Constant/copy propagation**
- **Dead store**
- **Redundant store**
Implementing SC with Compiler Support

• Idea: Implement a compiler pass (e.g., LLVM) to deal with non-SC preserving optimizations

L1: \( t = X \times 2 \)
L2: \( u = Y \)
L3: \( v = X \times 2 \)

\( \Rightarrow \)

L1: \( t = X \times 2 \)
L2: \( u = Y \)
L3: \( v = t \)
C3: if (X modified since L1)
L3: \( v = X \times 2 \)

SC Semantics

- SC does not guarantee data race freedom
- Not a strong memory model

Program semantics

```c
a++;
buffer[index]++;
```
Questions

• How would you implement an RMW instruction with SC?
• Are memory models only relevant in systems with support for caches?
• Is memory consistency not needed in presence of cache coherence?
• Do memory models only impact hardware design?
Hardware Memory Models
Characterizing Hardware Memory Models

- **Relax program order**
  - Store → Load, Store → Store, ...
  - Applicable to pairs of operations with different addresses

- **Relax write atomicity**
  - Read other core’s write early
  - Applicable to only cache-based systems

- **Relax both program order and write atomicity**
  - Read own write early
Possible Interleavings Under SC and TSO

<table>
<thead>
<tr>
<th>Core C1</th>
<th>Core C2</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1: x = NEW;</td>
<td>S2: y = NEW;</td>
<td>/* Initially, x = 0 &amp; y = 0*/</td>
</tr>
<tr>
<td>L1: r1 = y;</td>
<td>L2: r2 = x;</td>
<td></td>
</tr>
</tbody>
</table>
Total Store Order

- Allows reordering stores to loads
- Can read own write early, not other’s writes
- Conjecture: widely-used x86 memory model is equivalent to TSO
TSO Rules

a == b or a != b

- If $L(a) <_p L(b) \Rightarrow L(a) <_m L(b)$
- If $L(a) <_p S(b) \Rightarrow L(a) <_m S(b)$
- If $S(a) <_p S(b) \Rightarrow S(a) <_m S(b)$
- If $S(a) <_p L(b) \Rightarrow S(a) <_m L(b)$ /* Enables FIFO Write Buffer */

Every load gets its value from the last store before it to the same address
Support for FENCE Operations in TSO

If $L(a) \triangleleft_p FENCE \Rightarrow L(a) \triangleleft_m FENCE$

If $S(a) \triangleleft_p FENCE \Rightarrow S(a) \triangleleft_m FENCE$

If $FENCE \triangleleft_p FENCE \Rightarrow FENCE \triangleleft_m FENCE$

If $FENCE \triangleleft_p L(a) \Rightarrow FENCE \triangleleft_m L(a)$

If $FENCE \triangleleft_p S(a) \Rightarrow FENCE \triangleleft_m S(a)$

If $S(a) \triangleleft_p FENCE \Rightarrow S(a) \triangleleft_m FENCE$

If $FENCE \triangleleft_p L(a) \Rightarrow FENCE \triangleleft_m L(a)$
**Possible Outcomes with TSO**

<table>
<thead>
<tr>
<th>Core C1</th>
<th>Core C2</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1: ( x = \text{NEW;} )</td>
<td>S2: ( y = \text{NEW;} )</td>
<td>/* Initially, ( x = 0 ) &amp; ( y = 0 */</td>
</tr>
<tr>
<td>L1: ( r1 = x; )</td>
<td>L3: ( r3 = y; )</td>
<td></td>
</tr>
<tr>
<td>L2: ( r2 = y; )</td>
<td>L4: ( r4 = x; )</td>
<td>/* Assume ( r2 = 0 ) &amp; ( r4 = 0 */</td>
</tr>
</tbody>
</table>

**TABLE 4.3:** Can \( r1 \) or \( r3 \) be Set to 0?
Possible Outcomes with TSO

Program order (<p) of Core C1

S1: x = NEW; /* NEW */
L1: r1 = x; /* NEW */
L2: r2 = y; /* 0 */

Program order (<p) of Core C2

S2: y = NEW; /* NEW */
L3: r3 = y; /* NEW */
L4: r4 = x; /* 0 */

Outcome: (r2, r4) = (0, 0) and (r1, r3) = (NEW, NEW)
RMW in TSO

Load of a RMW **cannot** be performed until earlier stores are performed (i.e., exited the write buffer)

- Effectively drains the write buffer

Load requires read–write coherence permissions, not just read permissions

To guarantee atomicity, the cache controller may not relinquish coherence permission to the block between the load and the store
Relationship between SC and TSO

Correct?

TSO

SC

Correct?

SC

TSO
Partial Store Order (PSO)

• Allows reordering of store to loads and stores to stores
• Writes to **different** locations from the same processor can be pipelined or overlapped and are allowed to reach memory or other cached copies out of program order

• Can read own write early, not other’s writes
## Opportunities to Reorder Memory Operations

<table>
<thead>
<tr>
<th>Core C1</th>
<th>Core C2</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1: data1 = NEW; S2: data2 = NEW; S3: flag = SET;</td>
<td>L1: r1 = flag; B1: if (r1 ≠ SET) goto L1; L2: r2 = data1; L3: r3 = data2;</td>
<td>/* Initially, data1 &amp; data2 = 0 &amp; flag ≠ SET <em>/ /</em> spin loop: L1 &amp; B1 may repeat many times */</td>
</tr>
</tbody>
</table>

**TABLE 5.1:** What Order Ensures r2 & r3 Always Get NEW?
Reorder Operations Within a Synchronization Block

<table>
<thead>
<tr>
<th>Core C1</th>
<th>Core C2</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1: acquire(lock)</td>
<td>A2: acquire(lock)</td>
<td>/* Arbitrary interleaving of L1i’s &amp; S1j’s */</td>
</tr>
<tr>
<td>/* Begin Critical Section 1 */</td>
<td>/* Begin Critical Section 2 */</td>
<td>/* Handoff from critical section 1*/</td>
</tr>
<tr>
<td>Some loads L1i interleaved with some stores S1j</td>
<td>Some loads L2i interleaved with some stores S2j</td>
<td>/* To critical section 2*/</td>
</tr>
<tr>
<td>/* End Critical Section 1 */</td>
<td>/* End Critical Section 2 */</td>
<td>/* Arbitrary interleaving of L2i’s &amp; S2j’s */</td>
</tr>
<tr>
<td>R1: release(lock)</td>
<td>R2: release(lock)</td>
<td></td>
</tr>
</tbody>
</table>
Optimization Opportunities

Non-FIFO coalescing write buffer

Support non-blocking reads

• Hide latency of reads
• Use lockup-free caches and speculative execution

Simpler support for speculation

• Need not compare addresses of loads to coherence requests
• For SC, need support to check whether the speculation is correct
Relaxed Consistency Rules

If $L(a) <_p \text{FENCE} \Rightarrow L(a) <_m \text{FENCE}$

If $S(a) <_p \text{FENCE} \Rightarrow S(a) <_m \text{FENCE}$

If $\text{FENCE} <_p \text{FENCE} \Rightarrow \text{FENCE} <_m \text{FENCE}$

If $\text{FENCE} <_p L(a) \Rightarrow \text{FENCE} <_m L(a)$

If $\text{FENCE} <_p S(a) \Rightarrow \text{FENCE} <_m S(a)$
Relaxed Consistency Rules

Maintain TSO rules for ordering two accesses to the same address only

- If \( L(a) <_p L'(a) \Rightarrow L(a) <_m L'(a) \)
- If \( L(a) <_p S(a) \Rightarrow L(a) <_m S(a) \)
- If \( S(a) <_p S'(a) \Rightarrow S(a) <_m S'(a) \)

Every load gets its value from the last store before it to the same address
Correct Implementation under Relaxed Consistency

<table>
<thead>
<tr>
<th>Core C1</th>
<th>Core C2</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1: data1 = NEW;</td>
<td>L1: r1 = flag;</td>
<td>/* Initially, data1 &amp; data2 = 0 &amp; flag ≠ SET */</td>
</tr>
<tr>
<td>S2: data2 = NEW;</td>
<td>B1: if (r1 ≠ SET) goto L1;</td>
<td></td>
</tr>
<tr>
<td>F1: FENCE</td>
<td>F2: FENCE</td>
<td></td>
</tr>
<tr>
<td>S3: flag = SET;</td>
<td>L2: r2 = data1;</td>
<td>/* L1 &amp; B1 may repeat many times */</td>
</tr>
<tr>
<td></td>
<td>L3: r3 = data2;</td>
<td></td>
</tr>
</tbody>
</table>
Correct Implementation under Relaxed Consistency

**TABLE 5.4: Adding FENCEs for XC to Table 5.2’s Critical Section Program.**

<table>
<thead>
<tr>
<th>Core C1</th>
<th>Core C2</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>F11: FENCE</strong></td>
<td><strong>F11: FENCE</strong></td>
<td>/* Arbitrary interleaving of L1i’s &amp; S1j’s */</td>
</tr>
<tr>
<td>A11: acquire(lock)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>F12: FENCE</strong></td>
<td>F21: FENCE</td>
<td>/* Handoff from critical section 1*/</td>
</tr>
<tr>
<td>Some loads L1i interleaved</td>
<td></td>
<td>/* To critical section 2*/</td>
</tr>
<tr>
<td>with some stores S1j</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>F13: FENCE</strong></td>
<td>F22: FENCE</td>
<td></td>
</tr>
<tr>
<td>R11: release(lock)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>F14: FENCE</strong></td>
<td>F23: FENCE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>/* Arbitrary interleaving of L2i’s &amp; S2j’s */</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>F24: FENCE</td>
<td></td>
</tr>
</tbody>
</table>
Relaxed Consistency Memory Models

Weak ordering

• Distinguishes between data and synchronization operations
• A synchronization operation is not issued until all previous operations are complete
• No operations are issued until the previous synchronization operation completes
Correct Implementation under Relaxed Consistency

Which fences are needed to ensure correct ordering and visibility between C1 and C2?

<table>
<thead>
<tr>
<th>Core C1</th>
<th>Core C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>F11: FENCE</td>
<td>/* Arbitrary interleaving of L1i’s &amp; S1j’s */</td>
</tr>
<tr>
<td>A11: acquire(lock)</td>
<td>/* Handoff from critical section 1*/</td>
</tr>
<tr>
<td>F12: FENCE</td>
<td>/* To critical section 2*/</td>
</tr>
<tr>
<td></td>
<td>/* Arbitrary interleaving of L2i’s &amp; S2j’s */</td>
</tr>
<tr>
<td>Some loads L1i interleaved with some stores S1j</td>
<td>F21: FENCE</td>
</tr>
<tr>
<td>F13: FENCE</td>
<td>R22: release(lock)</td>
</tr>
<tr>
<td>R11: release(lock)</td>
<td>F23: FENCE</td>
</tr>
<tr>
<td>F14: FENCE</td>
<td>F24: FENCE</td>
</tr>
</tbody>
</table>
Relaxed Consistency Memory Models

Release consistency

• Distinguishes between **acquire and release synchronization operations**
• RCsc - maintains SC between synchronization operations
• Acquire → all, all → release, and sync → sync
Relaxed Consistency Memory Models

Why should we use them?

- Performance

Why should we not use them?

- Complexity
## Hardware Memory Models: One Slide Summary

<table>
<thead>
<tr>
<th>Relaxation</th>
<th>W → R Order</th>
<th>W → W Order</th>
<th>R → RW Order</th>
<th>Read Others’ Write Early</th>
<th>Read Own Write Early</th>
<th>Safety net</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC [16]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM 370 [14]</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>serialization instructions</td>
</tr>
<tr>
<td>TSO [20]</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RMW</td>
</tr>
<tr>
<td>PC [13, 12]</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td>√</td>
<td>RMW</td>
</tr>
<tr>
<td>PSO [20]</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td>√</td>
<td>RMW, STBAR</td>
</tr>
<tr>
<td>WO [5]</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td>synchronization</td>
</tr>
<tr>
<td>RCsc [13, 12]</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
<td>√</td>
<td>release, acquire, nsync, RMW</td>
</tr>
<tr>
<td>RCpc [13, 12]</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>release, acquire, nsync, RMW</td>
</tr>
<tr>
<td>Alpha [19]</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td>MB, WMB</td>
</tr>
<tr>
<td>RMO [21]</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td>various MEMBAR’s</td>
</tr>
<tr>
<td>PowerPC [17, 4]</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
<td>√</td>
<td>SYNC</td>
</tr>
</tbody>
</table>
Desirable Properties of a Memory Model

Hard to satisfy all three properties

• Programmability
• Performance
• Portability

Think of SC

Pros
• Intuitive
• Serializability of instructions

Cons
• No atomicity of regions
• Inhibits compiler transformations
• Almost all recent architectures violate SC
Programming Language
Memory Models
Data-Race-Free-0 (DRF0) Model

• Conceptually similar to Weak Ordering
• Assumes no data races
  • **No guarantees for racy programs**
• Allows many optimizations in the compiler and hardware
Language Memory Models

Developed much later

- Recent standardizations are largely driven by languages

Most are based on the DRF0 model

Why do we need one?

- Isn’t the hardware memory model enough?
C++ Memory Model

• Adaptation of the DRFO memory model
  • SC for data race free programs

• C/C++ simply ignore data races
  • No safety guarantees in the language
Catch-Fire Semantics in C++

```cpp
X* x = NULL;
bool done = false;

Thread T1
x = new X();
done = true;

Thread T2
if (done) {
    x->func();
}
```
Catch-Fire Semantics in C++

```cpp
X* x = NULL;
bool done = false;
x = new X();
done = true;
if (done) {
    x->func();
}
```
Memory Operations in C++

Synchronization
- Lock, unlock, atomic load, atomic store, atomic RMW

Data
- Load, Store
## Reordering of Memory Operations in C++

<table>
<thead>
<tr>
<th>Compiler reordering allowed for M1 and M2</th>
<th>M1 is a data operation and M2 is a read synchronization operation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>M1 is write synchronization and M2 is data</td>
</tr>
<tr>
<td></td>
<td>M1 and M2 are both data with no synchronization between them</td>
</tr>
<tr>
<td></td>
<td>M1 is data and M2 is the write of a lock operation</td>
</tr>
<tr>
<td></td>
<td>M1 is unlock and M2 is either a read or write of a lock</td>
</tr>
</tbody>
</table>
Writing Correct C++ Code

• Mutual exclusion of critical code blocks

```cpp
std::mutex mtx;
{
    mtx.lock();
    // access shared data here
    mtx.unlock();
}
```

• Mutex provides inter-thread synchronization
  • Unlock() synchronizes with calls to lock() on the same mutex object
Synchronize Using Locks

```cpp
std::mutex mtx;
bool dataReady = false;

{
    mtx.lock();
    prepareData();
    dataReady = true;
    mtx.unlock();
}

{
    mtx.lock();
    if (dataReady) {
        consumeData();
    }
    mtx.unlock();
}
```
Synchronize Using Locks

```cpp
std::mutex mtx;
bool dataReady = false;

{
    mtx.lock();
    prepareData();
    dataReady = true;
    mtx.unlock();
}

bool b;
{
    mtx.lock();
    b = dataReady;
    mtx.unlock();
}
if (b) {
    consumeData();
}
```
Using Atomics from C++11

• “Data race free” by definition
  • E.g., std::atomic<int>
  • A store synchronizes with operations that load the stored value
  • Similar to volatile in Java

• C++ volatile is different!
  • Does not establish inter-thread synchronization, not atomic
  • Can be part of a data race

```cpp
std::mutex mtx;
std::atomic<bool> ready(false);

prepareData();
ready.store(true);
if (ready.load()) {
  consumeData();
}

reg_var1 = reg_var2;

atm_var1.store(atm_var2.load());
```

```cpp
if (ready.load()) {
  consumeData();
}
```
Visibility and Ordering

Visibility

• When are the effects of one thread visible to another?

Ordering

• When can operations of any given thread appear out of order to another thread?
Ensuring Visibility

• Writer thread releases a lock
  • Flushes all writes from the thread’s working memory
• Reader thread acquires a lock
  • Forces a (re)load of the values of the affected variables
• Atomic (C++)/volatile (Java)
  • Values written are made visible immediately before any further memory operations
  • Readers reload the value upon each access
• Thread join
  • Parent thread is guaranteed to see the effects made by the child thread
Memory Order of Atomics

• Specifies how regular, non-atomic memory accesses are to be ordered around an atomic operation

• Default is sequential consistency

```c
atomic.h

enum memory_order {
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst
};
```
Memory Model Synchronization Modes

- Producer thread creates data
- Producer thread stores to an atomic
- Consumer threads read from the atomic
- When the expected value is seen, data from the producer thread is complete and visible to the consumer thread

The different memory model modes indicate how strong this data-sharing bond is between threads

http://gcc.gnu.org/wiki/Atomic/GCCMM/AtomicSync
Memory Model Modes

- `memory_order_seq_cst`

```c
x = 0;
y = 0;
y = 1;
x.store(2);
if (x.load() == 2)
    assert (y == 1)
```

Can this assert fail?
Memory Model Modes

- `memory_order_seq_cst`

```cpp
x = 0;
y = 0;
y.store(20);
x.store(10);
if (x.load() == 10)
    assert (y.load() == 20);
y.store(10);
if (y.load() == 10)
    assert (x.load() == 10)
```

Can these asserts fail?
Memory Model Modes

- `memory_order_relaxed`: no happens-before edges

```cpp
x = 0;
y = 0;
y.store(20, memory_order_relaxed);
x.store(10, memory_order_relaxed);
if (x.load(memory_order_relaxed) == 10)
    assert (y.load(memory_order_relaxed) == 20);
y.store(30, memory_order_relaxed);
if (y.load(memory_order_relaxed) == 30)
    assert (x.load(memory_order_relaxed) == 10)
```

Can these asserts fail?
Memory Model Modes

- memory_order_relaxed

```c
x = 0;
y = 0;
x.store(10, memory_order_relaxed);
x.store(20, memory_order_relaxed);
y = x.load(memory_order_relaxed);
z = x.load(memory_order_relaxed);
assert (y < z);
```

Can this assert fail?
Memory Model Modes

- `memory_order_acquire` and `memory_order_release`

```c
x = 0;
y = 0;
y.store(20, memory_order_release);
x.store(10, memory_order_release);
```

Can these asserts pass?

```c
assert (y.load(memory_order_acquire) == 20 && x.load(memory_order_acquire) == 0);
assert (y.load(memory_order_acquire) == 0 && x.load(memory_order_acquire) == 10);
```
Memory Model Modes

- `memory_order_acquire` and `memory_order_release`
Memory Model Modes

• memory_order_consume

n = 1;
m = 1;
p.store(&n, memory_order_release);

t = p.load(memory_order_acquire);
assert (*t == 1 && m == 1);

t = p.load(memory_order_consume);
assert (*t == 1 && m == 1);

Can these asserts fail?
Happens-Before Memory Model (HBMM)

• Read operation \( a = \text{rd}(t, x, v) \) may return the value written by any write operation \( b = \text{wr}(t, x, v) \) provided
  1. \( b \) does not happen after \( a \), i.e., \( b <_{HB} a \) or \( b \preceq a \)
  2. there is no intervening write \( c \) to \( x \) where \( b <_{HB} c <_{HB} a \)
x = 0;
y = 0;

y = 1;
r1 = x;

assert r1 != 0 || r2 != 0

x = 1;
r2 = y;

r1 = x;
y = 1;

assert r1 == 0 || r2 == 0

r2 = y;
x = 1;

Can these asserts fail?
Will the assertion pass or fail?
x = 10;

if (x != 0) {
    r2 = r1 / x;
}
x = 0;

Can anything go wrong?
• Potential for out-of-thin-air values
HBMM

- DRF0 allows arbitrary behavior for racy executions
  - DRF0 is not strictly stronger than HBMM

- HBMM does not guarantee SC for DRF programs
  - HBMM is not strictly stronger than DRF0
DRFO vs HBMM

```c
r1 = x;
if (r1 == 1)
y = 1;
```

```c
r2 = y;
if (r2 == 1)
x = 1;
```

assert r1 == 0 && r2 == 0

Is there a data race on x and y?
Java Memory Model (JMM)

• First high-level language to incorporate a memory model
• Provides memory- and type-safety, so has to define some semantics for data races
Outcomes Possible with JMM

• Racy Initialization

```
obj = new Circle();
if (obj != null)
    obj.draw()
```

Can there be a NPE with JMM?
Outcomes Possible with Java

```java
x = 0;
y = 0;
r1 = x;
y = 1;
r2 = y;
x = 1;
r2 = y;
```

```java
assert r1 != 0 || r2 != 0
```

```java
r1 = x;
y = 1;
r2 = y;
x = 1;
```

```java
assert r1 == 0 || r2 == 0
```
Outcomes Not Possible with Java

```java
r1 = x;
y = r1;
r2 = y;
x = r2;
```

assert r1 != 42

JMM is strictly stronger than DRF0 and HBMM
JVMs do not comply with the JMM!!!
Lessons Learnt

SC for DRF is the minimum baseline

- Make sure the program is free of data races
- System guarantees SC execution

Specifying semantics for racy programs is hard

Simple optimizations may introduce unintended consequences
References

• D. Sorin et al. A Primer on Memory Consistency and Cache Coherence
• C. Flanagan and S. Freund. Adversarial Memory for Detecting Destructive Races. PLDI 2010.