Memory Consistency Model

Swarnendu Biswas
UT Austin
Outline

- Data races
- Memory consistency models
- Sequential Consistency
- Hardware memory models
  - TSO, PSO, Relaxed consistency
- Language memory models
  - C++, Java
Today’s Trends
Data Race: Primary Source of Concurrency Errors

Object $X = \text{null}$;
$\text{boolean}\ \text{done} = \text{false}$;

Thread T1

$X = \text{new Object}()$;
$\text{done} = \text{true}$;

Thread T2

while (!\text{done}) {} 
$X.\text{compute}()$;
Object \( X = \text{null}; \)
\( \text{boolean \ done} = \text{false}; \)

**Thread T1**

```java
X = \text{new Object();}
done = \text{true;}
```

**Thread T2**

```java
while (\!\text{done}) {} 
X.\text{compute();}
```

---

**Data race**

**Conflicting accesses** – two threads access the same shared variable where at least one access is a write

**Concurrent accesses** – accesses are not ordered by synchronization operations
Thread T1

X = new Object();
done = true;

Thread T2

temp = done;
while (!temp) {}{ }

Infinite loop

Thread T1

done = true;
X = new Object();

Thread T2

while (!done) {}
X.compute();

NPE
Data Races are Bad

Therac-25 accident & Northeast US Blackout & NASDAQ Facebook glitch

---

**Technical Perspective**

Data Races are Evil with No Exceptions

By Sarita Adve

Exploiting parallelism has become the primary means to higher performance. Racy code. Java’s safety requirement preclude the use of “undefined” behavior.
**Memory Consistency Model: What Value Can a Read Return?**

**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>L2: Load r2 = data;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How a Core Might Reorder Accesses?

- Store-store
- Load-load
- Store-load
- Load-store
Memory Consistency Model

- Specifies the allowed behaviors of multithreaded programs executing with shared memory
  - Both at the hardware-level and at the programming-language-level

- “What values can a load return?”
  - Return the “last” write
  - Uniprocessor: program order
  - Multiprocessor: ?

- There can be multiple correct behaviors
Memory Consistency Model

• 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?
**TABLE 3.3:** Can Both r1 and r2 be Set to 0?

<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>

Dekker’s Algorithm
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
Earlier Example Under SC

program order (<p) of Core C1

S1: data = NEW; /* NEW */

S2: flag = SET; /* SET */

memory order (<m)

program order (<p) of Core C2

L1: r1 = flag; /* 0 */

L1: r1 = flag; /* 0 */

L1: r1 = flag; /* 0 */

L1: r1 = flag; /* SET */

L2: r2 = data; /* NEW */
SC Rules

• \( a = b \) or \( a \neq b \)
  • 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 \( S(a) <p L(b) \) \( \Rightarrow \) \( S(a) <m L(b) \)

• Every load gets its value from the last store before it (in global memory order) to the same address
Initially $A = B = 0$

P1    P2    P3
A = 1
if (A == 1)
    B = 1
    if (B == 1)
        register1 = A
Write Atomicity

- Relaxing write atomicity violates SC

<table>
<thead>
<tr>
<th></th>
<th>Initially X=Y=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>X=1</td>
</tr>
<tr>
<td>T2</td>
<td>Y=1</td>
</tr>
<tr>
<td>T3</td>
<td>r1=X fence r2=Y</td>
</tr>
<tr>
<td>T4</td>
<td>r3=Y fence r4=X</td>
</tr>
</tbody>
</table>

r1=1, r2=0, r3=1, r4=0 violates write atomicity
End-to-end SC

- Simple memory model that can be implemented both in hardware and in languages
- Performance
  - Naive hardware
    - Maintain program order - expensive for a write
      - E.g., write buffer can break Dekker’s algorithm
    - Write atomicity
- Program semantics
  - SC does not guarantee data race freedom
  - Not a strong memory model

```plaintext
a++;
buffer[index]++;
```
Cache Coherence

- Single writer multiple readers
- Memory updates are passed correctly, cached copies always contain the most recent data
- Virtually a synonym for SC

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*
Initially $A = B = C = 0$

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A = 1;$</td>
<td>$A = 2;$</td>
<td>while $(B \neq 1)$ ;</td>
<td>while $(B \neq 1)$ ;</td>
</tr>
<tr>
<td>$B = 1;$</td>
<td>$C = 1;$</td>
<td>while $(C \neq 1)$ ;</td>
<td>while $(C \neq 1)$ ;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\times$ tmp1 = $A$;</td>
<td>$\times$ tmp2 = $A$;</td>
</tr>
</tbody>
</table>
• Cache Coherence does not define shared memory behavior
  • Goal is to make caches invisible

• Memory consistency can use cache coherence as a “black box”
Characterizing Hardware Memory Models

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

- Relax write atomicity
  - Read own write early
  - Read other’s write early
    - Applicable to only cache-based systems
Read Other’s Write Early Can Violate Write Atomicity

Initially \( A = B = 0 \)
P1
A = 1
P2
while (A \(!=\) 1)
B = 1;
(P3)
tmp = A

P1
Write, A, 1
P2
Read, A, 1
Write, B, 1
P3
Read, B, 1
Read, A,
**TABLE 3.3: Can Both r1 and r2 be Set to 0?**

<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 (TSO)

- 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

• \( a \equiv b \) or \( a \neq b \)
  • 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

• Needs a notion of a FENCE
• If $L(a) < p \ FENCE \Rightarrow L(a) < m \ FENCE$
• If $S(a) < p \ FENCE \Rightarrow S(a) < m \ FENCE$
• If $FENCE < p \ FENCE \Rightarrow FENCE < m \ FENCE$
• If $FENCE < p \ L(a) \Rightarrow FENCE < m \ L(a)$
• If $FENCE < p \ S(a) \Rightarrow FENCE < m \ S(a)$

• If $S(a) < p \ FENCE \Rightarrow S(a) < m \ FENCE$
• If $FENCE < p \ L(a) \Rightarrow FENCE < m \ L(a)$
• Load of a RMW cannot be performed until earlier stores are performed (i.e., exited 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
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;</td>
<td></td>
<td>/* Initially, data1 &amp; data2 = 0 &amp; flag ≠ SET */</td>
</tr>
<tr>
<td>S2: data2 = NEW;</td>
<td></td>
<td>/* spin loop: L1 &amp; B1 may repeat many times */</td>
</tr>
<tr>
<td>S3: flag = SET;</td>
<td>L1: r1 = flag;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B1: if (r1 ≠ SET) goto L1;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L2: r2 = data1;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L3: r3 = data2;</td>
<td></td>
</tr>
</tbody>
</table>
### TABLE 5.2: What Order Ensures Correct Handoff from Critical Section 1 to 2?

<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 \( \text{FENCE} <^p S(a) \Rightarrow \text{FENCE} <^m S(a) \)

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></td>
<td>/* Initially, data1 &amp; data2 = 0 &amp; flag ≠ SET */</td>
</tr>
<tr>
<td>S2: data2 = NEW;</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>F1: FENCE</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S3: flag = SET;</td>
<td>L1: r1 = flag;</td>
<td>/* L1 &amp; B1 may repeat many times */</td>
</tr>
<tr>
<td></td>
<td>B1: if (r1 ≠ SET) goto L1;</td>
<td></td>
</tr>
<tr>
<td><strong>F2: FENCE</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>L2: r2 = data1;</td>
<td></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>F11: FENCE</td>
<td>F21: FENCE</td>
<td>/* Arbitrary interleaving of L1i’s &amp; S1j’s */</td>
</tr>
<tr>
<td>A11: acquire(lock)</td>
<td>A21: acquire(lock)</td>
<td>/* Handoff from critical section 1*/</td>
</tr>
<tr>
<td>F12: FENCE</td>
<td>F22: FENCE</td>
<td>/* To critical section 2*/</td>
</tr>
<tr>
<td>Some loads L1i interleaved with</td>
<td>Some loads L2i interleaved with</td>
<td>/* Arbitrary interleaving of L2i’s &amp; S2j’s */</td>
</tr>
<tr>
<td>some stores S1j</td>
<td>some stores S2j</td>
<td></td>
</tr>
<tr>
<td>F13: FENCE</td>
<td>F23: FENCE</td>
<td></td>
</tr>
<tr>
<td>R11: release(lock)</td>
<td>R22: release(lock)</td>
<td></td>
</tr>
<tr>
<td>F14: FENCE</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

• 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
<table>
<thead>
<tr>
<th>Hardware Memory Models: One Slide Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Relaxation</strong></td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>SC [16]</td>
</tr>
<tr>
<td>IBM 370 [14]</td>
</tr>
<tr>
<td>TSO [20]</td>
</tr>
<tr>
<td>PC [13, 12]</td>
</tr>
<tr>
<td>PSO [20]</td>
</tr>
<tr>
<td>WO [5]</td>
</tr>
<tr>
<td>RCsc [13, 12]</td>
</tr>
<tr>
<td>RCpc [13, 12]</td>
</tr>
<tr>
<td>Alpha [19]</td>
</tr>
<tr>
<td>RMO [21]</td>
</tr>
<tr>
<td>PowerPC [17, 4]</td>
</tr>
</tbody>
</table>
DRF0 Model

• Conceptually similar to WO
• Assumes no data races
• Allows many optimizations in the compiler and hardware
Language Memory Models

- Developed much later
- Most are based on the data-race-free-0 (DRF0) model

Why do we need one?
- Isn’t the hardware memory model enough?
• Adaptation of the DRFO memory model
  • SC for data race free programs

• C/C++ simply ignore data races
  • No safety guarantees in the language

• Memory operation
  • Synchronization: lock, unlock, atomic load, atomic store, atomic RMW
  • Data: Load, Store
• Compiler reordering **allowed** for memory operations M1 and M2 when:

  • M1 is a data operation and M2 is a read synchronization operation
  • M1 is write synchronization and M2 is data
  • M1 and M2 are both data with no synchronization between them
  • M1 is data and M2 is the write of a lock operation
  • M1 is unlock and M2 is either a read or write of a lock
Write Correct C++ Code

• Mutually exclusive execution 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
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;

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

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

• “Data race free” variable by definition: `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> dataReady(false);

prepareData();
if (dataReady.load()) {
  consumeData();
}
```
Memory Order of Atomics

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

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
};
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?
Relaxed Ordering

// Thread 1:
r1 = y.load(memory_order_relaxed);
x.store(r1, memory_order_relaxed);

// Thread 2:
r2 = x.load(memory_order_relaxed); // C
y.store(42, memory_order_relaxed); // D

Is r1 == r2 == 42 possible?
Is \( r_1 = r_2 = 42 \) possible?
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
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

Initially \( x = y = 0 \)

Thread 1:

\[
\begin{align*}
y & = 1; \\
r_1 & = x;
\end{align*}
\]

Thread 2:

\[
\begin{align*}
x & = 1; \\
r_2 & = y;
\end{align*}
\]

assert \( r_1 \neq 0 \) \(|\) \( r_2 \neq 0 \)
Initially $x = y = 0$

Thread 1:

$r1 = x$;
$y = 1$;

Thread 2:

$r2 = y$;
$x = 1$;

assert $r1 == 0 || r2 == 0$
Initially $x = 0$

Thread 1:

$x = 7$;

Thread 2:

$\text{if } (x \neq 0)$

$r2 = r1 \, / \, x;$
Initially $x = y = 0$

Thread 1:

$r1 = x$

if ($r1 == 1$)

    $y = 1$;

assert $r1 == 0$ && $r2 == 0$

Thread 2:

$r2 = y$

if ($r2 == 1$)

    $x = 1$;
Initially \( x = y = 0 \)

Thread 1:

\[
\begin{align*}
    r1 & = x; \\
    y & = r1;
\end{align*}
\]

Thread 2:

\[
\begin{align*}
    r2 & = y; \\
    x & = r2;
\end{align*}
\]

assert \( r1 \neq 42 \)
Initially $x = y = 0$

Thread 1:

1. $r1 = x$;
2. $y = r1$;

Thread 2:

3. $r2 = y$;
4. if ($r2 == 1$) {
5. \hspace{1em} $r3 = y$;
6. \hspace{1em} $x = r3$;
7. \hspace{1em} } else $x = 1$;

assert $r2 == 0$
What Constitutes a Good Memory Model?

- Programmability
- Performance
- Portability
- Precision
Lessons Learnt

• SC for DRF is the minimal 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
Memory Consistency Model

Swarndenu Biswas
UT Austin