C++ Memory Model
Motivation and Explanation of the Model

Martin Kempf
University of Applied Sciences Rapperswil (HSR), Switzerland

8/1/2013
Master Seminar: Program Analysis and Transformation
Outline

• Strong SW Model – Weak HW Model
• Motivation for a Memory Model in C++
• C++ Memory Model
  – Memory Model Basics
  – Sequentially Consistent Atomics
  – Low-Level Atomics introduced
• Conclusion
Hardware Memory Models are weaker than SW Memory Models

- Hardware Memory Models are more relaxed

```c
if (IsPublished)
{
    LOADLOAD_FENCE(); // Prevent reordering of loads
    return Value; // Load published value
}
```

- Software Memory Model are stronger as more important for usability
- Mapping for optimal performance is difficult.
- Additionally, Software is mapped on different Hardware with different models.
Pthread, a working but non solid solution for threading prior C++11

- prior to C++11 the C++ standard didn't provide a memory model aware of concurrent execution of program code.
- Threading with Pthread works but:
  - relies on the compiler.
  - No guaranteed portability.
Performance and portability motivated for a C++ Memory Model

• Portability
  – Same code - different compiler - different compiler optimizations – different hardware
    → Same behavior

• Performance
  – Simple usage of atomic instructions
  – No locking for atomic execution of e.g. increment

lock()  x++;  unlock()  x.atomicIncrement()
The C++ Memory Model offers different Ordering Options

- Default model is sequential consistent
- Undefined behavior in case of data race
Objects and Memory Location define conflicting variables

- Two threads of execution can update and access separate memory locations without interfering with each other.
Modification order is agreed by all threads

<table>
<thead>
<tr>
<th>Programs</th>
<th>Possible Execution</th>
<th>Modification Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>prog1</td>
<td>thread1</td>
<td>Var A</td>
</tr>
<tr>
<td>1. A.store(3)</td>
<td>1. A.store(3)</td>
<td>1.</td>
</tr>
<tr>
<td>2. B.store(4)</td>
<td>2. B.store(4)</td>
<td>5.</td>
</tr>
<tr>
<td>4. B.load</td>
<td>4. B.load</td>
<td>7. B.load</td>
</tr>
</tbody>
</table>
Sequenced-before defines intra-thread order

```cpp
#include <iostream>

int subtract(int a, int b) {
    return a - b;
}

int get_num() {
    static int i = 0;
    i++; //unsequenced side effect resulting in undefined behaviour
    return i;
}

void main() {
    int sub = subtract(get_num(), get_num());
    std::cout << sub;
}
```
Conflicting, adjacent operations form a data race in SC

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

Thread 2
\[
\begin{align*}
y &= 1; \\
r_2 &= x;
\end{align*}
\]

Allows: \(r_1 = r_2 = 0\).

Data Race:
- same memory location (y)
- at least one is a store, atomic store or atomic rmw
- at least one is a data operation
- adjacent operations by different threads in interleaving
Sequential consistency for data-race free programs

```c
atomic<int> x, y; // initially zero
```

Thread 1
```
x = 1;
r1 = y;
```

Thread 2
```
y = 1;
r2 = x;
```

- **Disallows** \( r_1 = r_2 = 0 \).
- Compiler and hardware do whatever it takes.
- A program is data-race-free (on a particular input) if no sequentially consistent execution results in a data race.
New data race definition with \textit{happens-before}

Example program:

\begin{verbatim}
atomic<int> x = 0;
atomic<int> y = 0;

x.store(1, seq_cst);
y.store(1, seq_cst);
y.load(seq_cst);
x.load(seq_cst);
\end{verbatim}

Happens-before Relations:

\begin{itemize}
  \item \textit{W}_{SC} x=1 \quad \Rightarrow \quad \textit{R}_{SC} y=0
  \item \textit{W}_{SC} y=1 \quad \Rightarrow \quad \textit{R}_{SC} x=1
  \item \textit{h}b \quad \text{between operations}
\end{itemize}

The execution contains a \textbf{data race} if two operations:
\begin{itemize}
  \item Access the same location by different threads
  \item One is not an atomic operation
  \item Neither happens-before the other
\end{itemize}
Relaxed Ordering

Example program:

```cpp
atomic<int> x = 0;
atomic<int> y = 0;

x.store(1, relaxed);
y.store(1, relaxed);

y.load(relaxed);
x.load(relaxed);
```

**Happens-before** Relations:

- $W_{RLXX} = 1$
- $W_{RLXY} = 1$
- $R_{RLXY} = 0$
- $R_{RLXX} = 0$

Although read of $x = 0$ and $y = 0$ there is **no data race** because the conflicting operations are both atomic.
**Happens-before** in Acquire-Release Ordering

Example program:

```c
int x = 0;
atomic<int> y = 0; //sender
//x = ...
y.store(1, release); //receiver
while (0 == y.load(acquire))
r = x;
```

Relations:
Release sequence in synchronizes-with

Example program:

```cpp
int x = 0;
atomic<int> y = 0;
//sender
x = …
y.store(1, release);
y.store(2, relaxed);
```

```
//receiver
while (0 == y.load(acquire))
r = x;
```

Relations:

Visual sequence of side effects – possible reads:
Relaxing Acquire-Release Ordering with Data Dependency

Example program:

```
atomic<int*> p; atomic<int> a;
int data;
//sender a.store(99, relaxed); //receiver
data = 1;
p.store(&data, release));
```

```
int* r1;
r1 = p.load(consume); //read data
*r1; //deref. data
```

```
a.load(relaxed) == 99;
```

Relations:
Conclusion

• C++ memory model provides:
  – a guarantee for portable code across platforms.
  – Sequential consistency if program is data-race-free and does not use low-level atomics.
  – Atomics for lock-free algorithms with increased performance.
  – Low-level atomics for more care in multi-core performance. (but also more difficult to use)