The C/C++ Memory Model: Overview and Formalization

Mark Batty  Jasmin Blanchette  Scott Owens
Susmit Sarkar  Peter Sewell  Tjark Weber

UPMARC
Verification of Concurrent C Programs
C11 / C++11

In 2011, new versions of the ISO standards for C and C++, informally known as C11 and C++11, were ratified.

These standards define a memory model for C/C++.

Support for this model has recently become available in popular compilers (GCC 4.4, Intel C++ 13.0, MSVC 11.0, Clang 3.1).
Memory Models

A memory model describes the interaction of threads through shared data.
Sequential Consistency

“The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.”

Example (Dekker’s algorithm)

\[
\begin{align*}
    x &= 0; \ y &= 0; \\
    x &= 1; \quad & y &= 1; \\
    r1 &= y; \quad & r2 &= x; \\
    \text{assert} \ (r1 == 1 \ || \ r2 == 1);
\end{align*}
\]
Real Hardware

Real hardware doesn’t run the code that you wrote.
Concurrency in C/C++

- Pthreads
- Hardware model
- C11/C++11
Simple concurrency:
- Sequential consistency for data-race free code (→ locks).
- Data races cause undefined behavior.

Expert concurrency:
- Atomic memory locations
Data Races

- Two (or more) threads concurrently\(^1\) access the same memory location.
- At least one of the threads writes.

Example (Dekker’s algorithm)

```
int x(0); int y(0);

x = 1;  ||  y = 1;

r1 = y;  ||  r2 = x;
```

\(^1\)I.e., not ordered by happens-before.
Data Races

- Two (or more) threads concurrently\(^1\) access the same memory location.
- At least one of the threads writes.

Example (Dekker’s algorithm)

\[
\begin{align*}
\text{int } x(0); & \quad \text{int } y(0); \\
x & = 1; \quad \text{y} \quad \text{y} = 1; \\
r1 & = y; \quad \text{r2} \quad \text{r2} = x;
\end{align*}
\]

\(^1\)I.e., not ordered by happens-before.
Data Races

- Two (or more) threads concurrently\(^1\) access the same memory location.
- At least one of the threads writes.

Example (Dekker’s algorithm)

\[
\begin{align*}
\text{int } x(0); & \quad \text{int } y(0); \\
\text{x } = 1; & \quad \text{y } = 1; \\
r1 = y; & \quad r2 = x;
\end{align*}
\]

\(^1\)I.e., not ordered by happens-before.
Data Races

- Two (or more) threads concurrently\(^1\) access the same memory location.
- At least one of the threads writes.

\(^1\)I.e., not ordered by happens-before.
std :: atomic<T>

Operations:
- `x.load(memory_order)
- `x.store(T, memory_order)

Concurrent accesses on atomic locations do not race.¹

The memory_order argument specifies ordering constraints between atomic and non-atomic memory accesses in different threads.

¹Except during initialization.
std::memory_order

- strict
  - seq_cst
  - release/acquire
  - release/consume

- relaxed
  - no synchronization

{message passing

- total order (SC for DRF code)
There is a total order over all seq_cst operations. This order contributes to inter-thread ordering constraints.\(^1\)

**Example (Dekker’s algorithm)**

```c
atomic_int x(0); atomic_int y(0);
x. store (1, seq_cst );
y. store (1, seq_cst );
int r1 = y.load( seq_cst );
int r2 = x.load( seq_cst );
assert (r1 == 1 || r2 == 1);
```

\(^1\)Similar to memory_order_{release| acquire}.\)
std :: memory_order_release/acquire

An acquire load makes prior writes to other memory locations made by the thread that did the release visible in the loading thread.

Example (message passing)

```c
int data(0); atomic_bool flag(false);

// sender
data = 42;
flag.store(true, release);

// receiver
while (!flag.load(acquire))
{
    assert(data == 42);
}
```
A consume load makes prior writes to data-dependent memory locations made by the thread that did the release visible in the loading thread.

Example (message passing)

```cpp
int data(0); atomic<int*> p(0);

// sender
data = 42;
p. store(&data, release );

// receiver
while (p.load(consume) == 0)
    {}
assert(*p == 42);
```
Relaxed operations impose very weak inter-thread ordering constraints ($\rightarrow$ coherence).

Example (message passing)

```cpp
int data(0); atomic<int*> p(0);

// sender
data = 42;
p. store(&data, release );
// receiver
while (p.load(consume) == 0)
while (p.load( relaxed ) == 0) ;
assert (*p == 42);
```
Program executions consist of memory actions. The program source determines several relations over these actions.

Example

```c
int x = 0;
int y = (x == x);
```

```
W x = 0
sb
R x = 0
sb

R x = 0
sb
W y = 1
sb
```
A candidate execution is specified by three relations:

- **sc** is a total order over all seq_cst actions.
- **reads-from (rf)** relates write actions to read actions at the same location that read the written value.
- For each atomic location, the **modification order (mo)** is a total order over all writes at this location.

From these, various other relations (e.g., happens-before) are derived. The memory model imposes constraints on these relations.
Coherence

The following are all forbidden.

CoRR

CoWW

CoWR

CoRW
Consider all *consistent* candidate executions.

If at least one of them has a *data race*, the program has undefined behavior.

Otherwise, its semantics is the set of consistent candidate executions.

\(^{2}\) There are actually several kinds.
Fine Points

- Fences

- Self-satisfying conditionals

- DRF in a SC semantics, but not DRF in C(++)11

```cpp
atomic_int x(0); atomic_int y(0);
if (x.load(seq_cst) == 1) if (y.load(seq_cst) == 1)
  atomic_init(&y,1); atomic_init(&x,1);`
```
CppMem & Nitpick

source code

static semantics

consistent executions
Conclusion and Future Challenges

Since 2011, C and C++ have a memory model.

We have a formal (machine-readable, executable) version of this memory model.

- Compiler correctness
- Program transformations
- Static analysis
- Dynamic analysis
- Program logics
- Formal verification
- Equivalent models