C/C++ Concurrency: Formalization and Model Finding

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

7th International Workshop on Reachability Problems (RP'2013)
Informal Presentation

Concurrency in C/C++ (Before 2011)

- Pthreads 
- Hardware model 

C11 / C++11

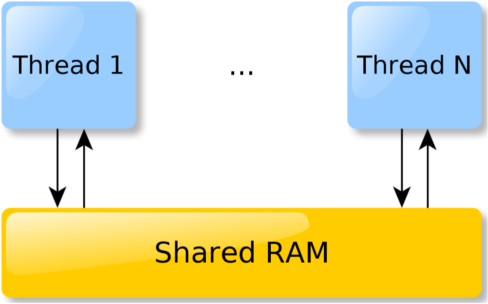
New versions of the ISO standards for C and C++ were ratified in 2011.

These standards define a memory model for C/C++ that allows programmers to write portable, yet highly efficient concurrent code.

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

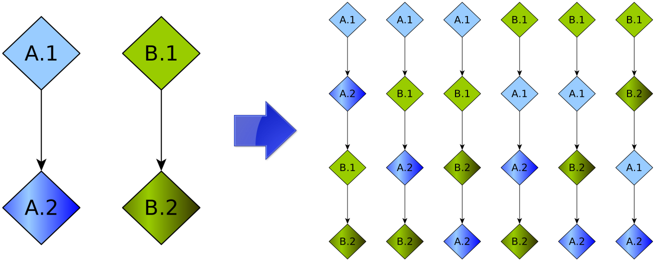
Memory Models

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."



C11/C++11 Concurrency

Simple concurrency:

- Sequential consistency for data-race free code ($\rightarrow$ 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)

```
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)

```
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.

[1] I.e., not ordered by happens-before.

std::atomic<T>

Operations:

- x.load(memory_order)
- x.store(T, memory_order)

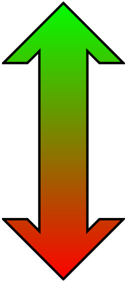
Concurrent accesses on atomic locations do not race.[1]

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

[1] Except during initialization.

std :: memory_order

strict



relaxed

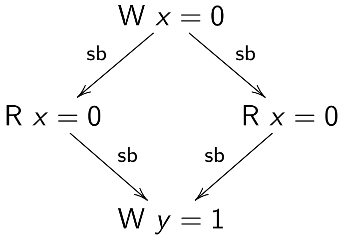
seq_cst	total order (SC for DRF code)
release / acquire	message passing
release / consume	message passing
relaxed	no synchronization

The Formal Model (1)

Program executions consist of memory actions. The program source determines several relations over these actions.

Example

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



The Formal Model (2)

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.

Consistent Executions

The memory model imposes constraints on these relations.

$$\textbf{consistent}\ \overbrace{\text{acts thrs lk sb asw dd cd}}^{\text{program}}\ \underbrace{\text{rf mo sc}}_{\text{execution}}\ \equiv$$

$$\vdots$$

Program Semantics

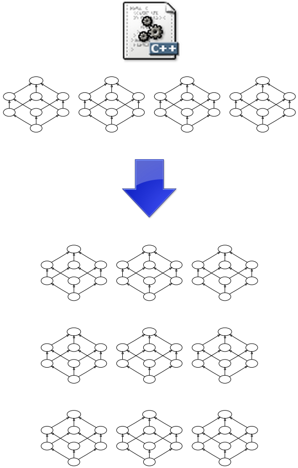
Consider all consistent candidate executions.

If at least one of them has a data race,[2] the program has undefined behavior.

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

[2] There are actually several kinds.

Exploring Program Behavior



source code

static semantics

consistent executions

Cppmem

We have generated an executable (OCaml) version of the **consistent** predicate from the formal memory model.

Cppmem exhaustively enumerates all candidate executions (for a given program) and tests for consistency.

Nitpick

Nitpick is a generic model finder for higher-order logic.

Nitpick translates the formal memory model—together with the constraints imposed by a given program—into first-order relational logic ($\to$ Kodkod) and then into SAT.

Example: Write-to-Read Causality

```
atomic_int x = 0, y = 0;

{{{
    x.store(1, rlx);
|||
    x.load(rlx);          // = 1
    y.store(1, rlx);
|||
    y.load(rlx);          // = 1
    x.load(rlx);          // = 0
}}}
```

Write-to-Read Causality: Performance

Locations (n)	Actions ($3n+1$)	States $\left(2^{3(3n+1)^2}\right)$	CPPMEM relaxed	CPPMEM SC	NITPICK relaxed	NITPICK SC
2	7	2^{147}	0.0 s	0.5 s	4 s	4 s
3	10	2^{300}	0.0 s	90.5 s	11 s	11 s
4	13	2^{507}	0.1 s	$>10^4$ s	41 s	40 s
5	16	2^{768}	0.2 s		132 s	127 s
6	19	2^{1083}	0.7 s		384 s	376 s
7	22	2^{1452}	2.5 s		982 s	977 s

Conclusion

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

We have

- a formal (machine-readable, executable) version of this memory model and
- automatic tools to explore the behavior of small programs.

Future Challenges

Use the model!

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