CppMem: An Overview

This lesson gives a brief overview of CppMem and how it helps in optimization.

WE'LL COVER THE FOLLOWING

CppMem is an interactive tool for exploring the behavior of small code snippets using the C++ memory model. It has to be in the toolbox of each programmer who seriously deals with the memory model.

The online version of CppMem - you can also install it on your PC - provides very valuable services in a twofold way:

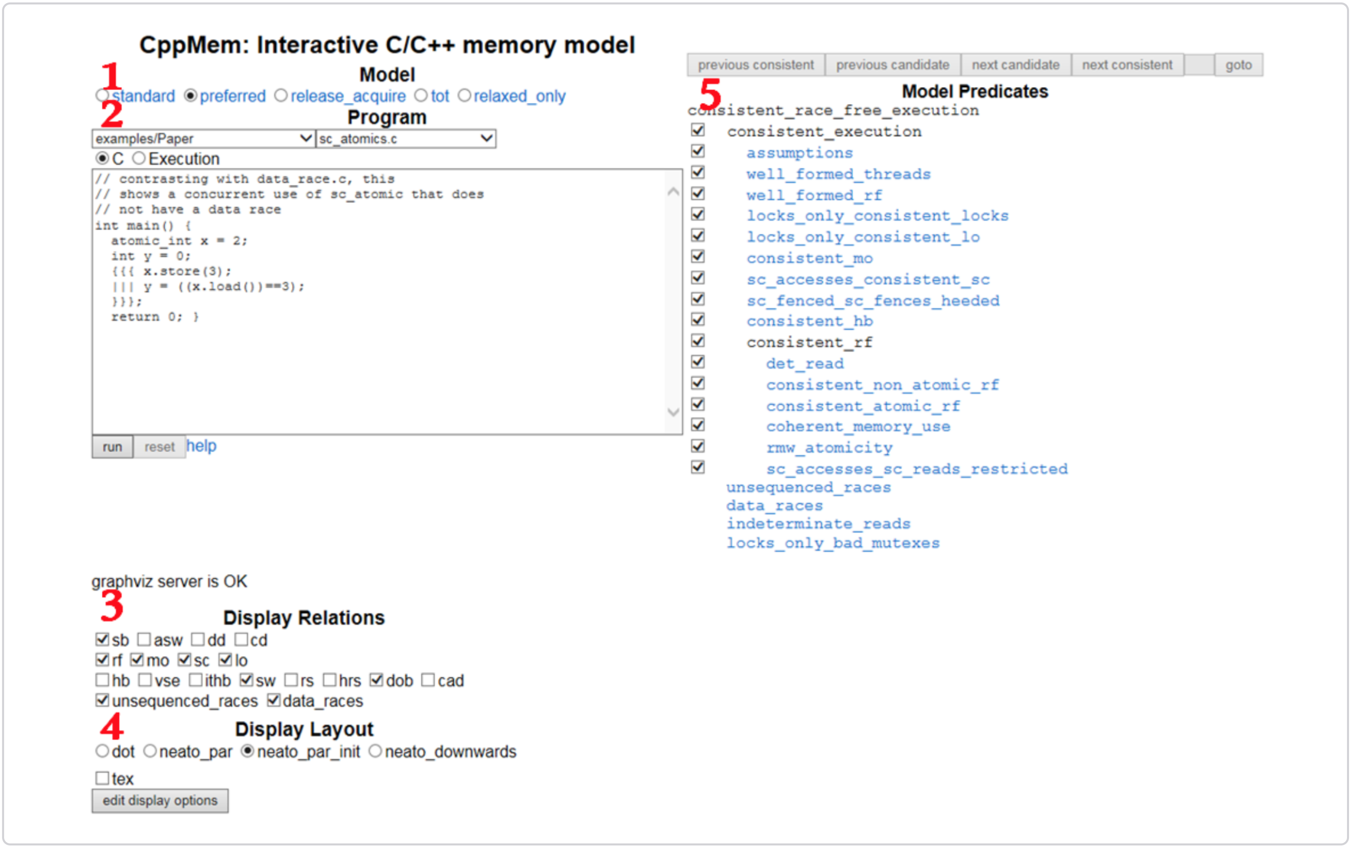
1. CppMem verifies the behavior of small code snippets. Based on the C++ memory model, the tool considers all possible interleavings of threads, visualizes each of them in a graph, and annotates these graphs with additional details.
2. The very accurate analysis of CppMem gives you deep insight into the C++ memory model. In short, CppMem is a tool that helps you to get a better understanding of the memory model.

Of course, it's often the nature of powerful tools that you first have to overcome a few hurdles. The nature of things is that CppMem is highly configurable, and it gives you a very detailed analysis related to this

extremely challenging topic. Therefore, my plan is to present the components of the tool.

The Overview

My simplified overview of CppMem is based on the default configuration. This overview will only provide you with the base for further experiments and should help you to understand my process of ongoing optimization.


CppMem: Interactive C/C++ memory model

1
Model
standard [✓ preferred] release_acquire tot relaxed_only

2
Program
examples/Paper | sc_atomics.c
[✓ C] Execution

```
// contrasting with data_race.c, this
// shows a concurrent use of sc_atomic that does
// not have a data race
int main() {
  atomic_int x = 2;
  int y = 0;
  {{{ x.store(3);
  ||| y = ((x.load())==3);
  }}};
  return 0; }
```

graphviz server is OK

3
Display Relations
[✓ sb] asw dd cd
[✓ rf] [✓ mo] [✓ sc] [✓ lo]
hb vse ithb [✓ sw] rs hrs [✓ dob] cad
[✓ unsequenced_races] [✓ data_races]

4
Display Layout
dot neato_par [✓ neato_par_init] neato_downwards
tex

5
Model Predicates

```
consistent_race_free_execution
✓ consistent_execution
✓   assumptions
✓   well_formed_threads
✓   well_formed_rf
✓   locks_only_consistent_locks
✓   locks_only_consistent_lo
✓   consistent_mo
✓   sc_accesses_consistent_sc
✓   sc_fenced_sc_fences_heeded
✓   consistent_hb
✓   consistent_rf
✓     det_read
✓     consistent_non_atomic_rf
✓     consistent_atomic_rf
✓     coherent_memory_use
✓     rmw_atomicity
✓     sc_accesses_sc_reads_restricted
  unsequenced_races
  data_races
  indeterminate_reads
  locks_only_bad_mutexes
```

For the sake of simplicity, I will refer to the red numbers in the screenshot.

1. Model #
   - Specifies the C++ memory model. *preferred* is a simplified variant of the C++11 memory model.

2. Program #
   - Contains the executable program (e.g. syntax) in a simplified *C or C++.

To be precise, you cannot directly copy *C or C++* code programs into CppMem.

- You can choose between a lot of programs that implement typical multithreading scenarios. To get the details of these programs, read the very well written article Mathematizing C++ Concurrency. Of course, you can also run your own code.
- CppMem is about multithreading; therefore, there are shortcuts.
  - You can easily define two threads using the expression `{{{ ... ||| ... }}}`. The three dots `(...)` represent the work package of each thread.
  - If you use the expression `x.readvalue(1)`, CppMem will evaluate the interleavings of the threads for which the thread execution gives the value `1` for `x`.

3. Display Relations

- Describes the relations between the read, write, and read-write modifications on atomic operations, fences, and locks.
- You can explicitly enable the relations in the annotated graph with the switches.
- There are three classes of relations. The coarser distinction between original and derived relations is the most interesting one. Here are the default values.
  - Original relations:
    - **sb**: sequenced-before
    - **rf**: read from
    - **mo**: modification order
    - **sc**: sequentially consistent
    - **lo**: lock order
  - Derived relations:
    - **sw**: synchronises-with
    - **dob**: dependency-ordered-before
    - **unsequenced_races**: races in a single thread

- **data_races**: inter-thread data races

4. Display Layout
   - With this switch, you can choose which Doxygraph graph is used.

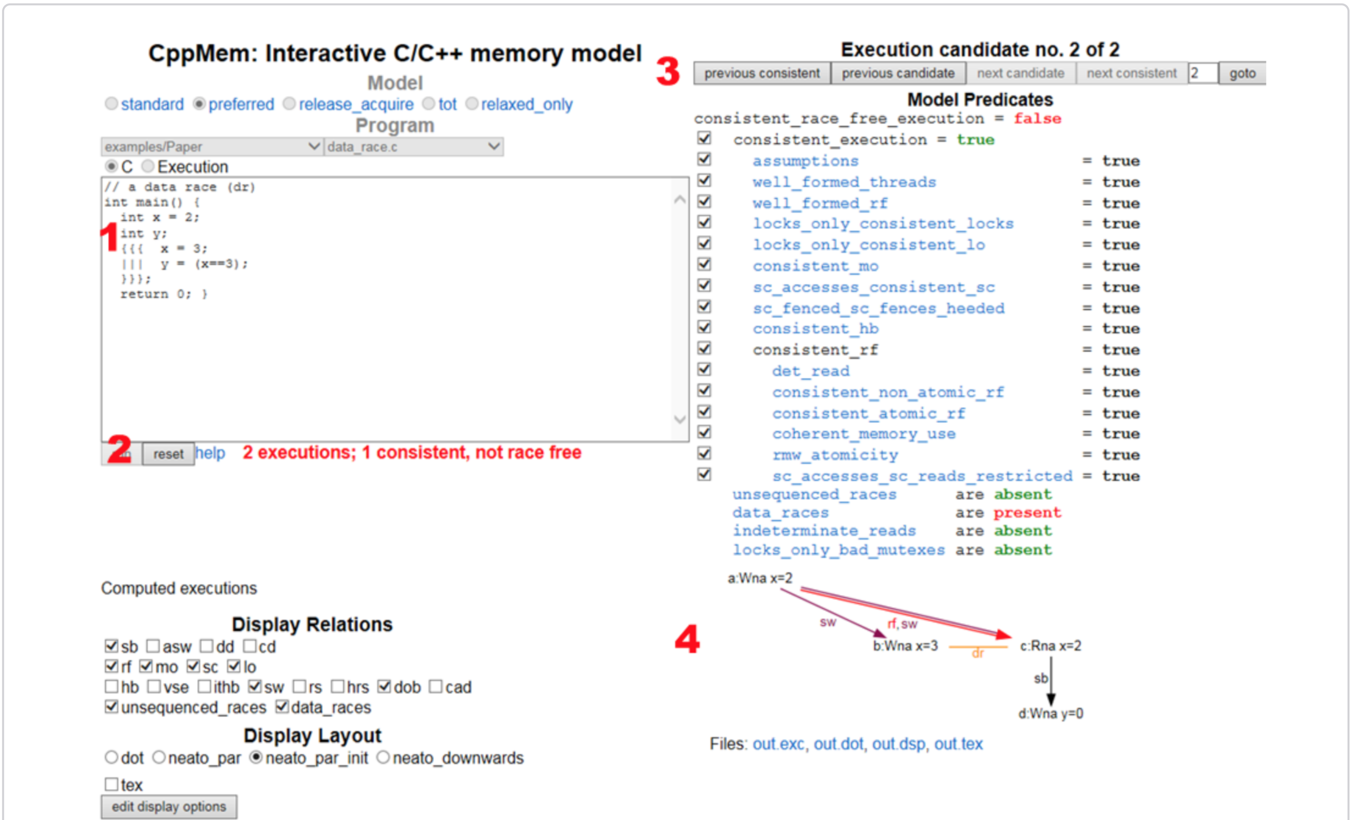
5. Model Predicates
   - With this button, you can set the predicates for the chosen model - which can cause a non-consistent (not data-race-free) execution; therefore, if you get a non-consistent execution, you see exactly the reason for the non-consistent execution. I will not use this button in this course.

See the documentation for more details.

This is enough as a starting point for my ongoing optimization. Now, it is time to give CppMem a try.

The Test Run

You have to choose the program `data_race.c` from the CppMem samples. The run button shows immediately that there is a *data race*.



CppMem: Interactive C/C++ memory model

Model

standard [✓ preferred] release_acquire tot relaxed_only

Program

examples/Paper data_race.c

[✓ C] Execution

```
// a data race (dr)
int main() {
  int x = 2;
  int y;
  {{{  x = 3;
  |||  y = (x==3);
  }}};
  return 0; }
```

2 executions; 1 consistent, not race free

Computed executions

Display Relations

☑sb ☐asw ☐dd ☐cd
☑rf ☑mo ☑sc ☑lo
☐hb ☐vse ☐ithb ☑sw ☐rs ☐hrs ☑dob ☐cad
☑unsequenced_races ☑data_races

Display Layout

dot neato_par [✓ neato_par_init] neato_downwards

☐tex

Execution candidate no. 2 of 2

Model Predicates

```
consistent_race_free_execution = false
☑ consistent_execution = true
☑   assumptions                        = true
☑   well_formed_threads                = true
☑   well_formed_rf                     = true
☑   locks_only_consistent_locks        = true
☑   locks_only_consistent_lo           = true
☑   consistent_mo                      = true
☑   sc_accesses_consistent_sc          = true
☑   sc_fenced_sc_fences_heeded         = true
☑   consistent_hb                      = true
☑   consistent_rf                      = true
☑     det_read                         = true
☑     consistent_non_atomic_rf         = true
☑     consistent_atomic_rf             = true
☑     coherent_memory_use              = true
☑     rmw_atomicity                    = true
☑     sc_accesses_sc_reads_restricted = true
  unsequenced_races       are absent
  data_races              are present
  indeterminate_reads     are absent
  locks_only_bad_mutexes  are absent
```

Files: out.exc, out.dot, out.dsp, out.tex

Explanation:

1. The data race is quite easy to see. A thread writes `x (x = 3)` and another thread reads `x (x==3)` without synchronization.
2. Two interleavings of threads are possible due to the C++ memory model; only one of them is consistent to the chosen model. This is the case if, in the expression `x==3`, the value of x is written by the expression `int x = 2` in the main function. The graph displays this relation in the edge annotated with rf and sw.
3. It is extremely interesting to switch between the different interleaving of threads.
4. The graph shows all relations in the format display layout, which you enabled in the Display Relations.

- `a:Wna x=2` is in the graphic as the a-th statement, which is a non-atomic write. `Wna` stands for “Write non-atomic”.
- The key edge in the graph is the edge between the writing of `x (b:Wna)` and the reading of `x (C:Rna)`. That’s the data race on `x`.

Let’s move on to the non-atomic variables in the next lesson.