Automatically Comparing Memory Consistency Models

John Wickerson
Imperial College London

Mark Batty
University of Kent

Tyler Sorensen
Imperial College London

George A. Constantinides
Imperial College London

Context

In the specification of languages and architectures, a **memory consistency model** (MCM) defines what happens when threads access **shared memory locations**, and the extent to which different threads see consistent data. MCMs often take the form of a **set of axioms** that characterise which of a program's executions are allowed.

Multiprocessors (x86, ARM, Power), graphics processors (Nvidia, AMD), and high-level languages (C, C++, OpenCL) all define their own MCM.

Problem

Because MCMs take into account various optimisations employed by compilers and architectures, they are often **complex and counterintuitive**. This makes them challenging to design and to understand. Indeed, many bugs have been traced back to programmers, compiler-writers, and architects misunderstanding MCMs.

In particular,

- it is hard to assess the impact of a **proposed change** to an MCM's set of axioms,
- it is hard to ensure that a **compiler mapping** correctly implements its source language's MCM using its target language's MCM, and
- it is hard to ensure that a **compiler optimisation** is valid for a given MCM.

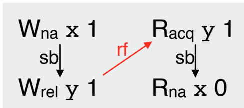
Our solution

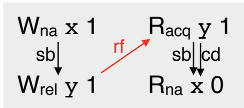
We use a constraint solver called **Alloy** to search for a program execution within the 'diff' between two given MCMs, gradually increasing the upper bound on execution size until one is found or time runs out. We then construct a litmus test that can only pass by taking this execution. (Getting Alloy to generate litmus tests directly is computationally infeasible.)

- If we give Alloy two variants of the same MCM, then any resultant litmus test is **minimal for distinguishing them**.
- If we give Alloy a source MCM, and a target MCM composed with a compiler mapping, then any resultant litmus test is a **minimal example of a bug in the mapping**.
- If we give Alloy the same MCM twice, the second copy composed with a compiler optimisation, then any resultant litmus test is a **minimal example of a bug in the optimisation**.

A thorny issue

How can we avoid constructing C++ litmus tests that are **racy** (and hence useless)? For example, these two executions are both disallowed by C++ ...





... and give rise (respectively) to similar-looking litmus tests...

```
x=1;          || r0=load(y,
store(y,1,    ||   ACQ);
  REL);       || r1=x;
```

```
x=1;          || r0=load(y,
store(y,1,    ||   ACQ);
  REL);       || if(r0) r1=x;
```

... but the left-hand test is racy! We avoid this problem by not generating executions like the top-left one in the first place. We achieve this by imposing an extra constraint, called **deadness**. (See our paper for more details.)

Results

- We compared **changes to the C++ MCM** proposed by Batty et al. (2016), Nienhuis et al. (2016), and Lahav et al. (2017) against the original C++ MCM. The distinguishing litmus tests that Alloy found automatically are **simpler than or the same as** those found manually by the respective authors.
- We checked some **compiler optimisations** against the C++ MCM, and found bugs that are **simpler than or the same as** those found manually by Vafeiadis et al. (2015).
- We checked **compiler mappings** from C++ to Power multiprocessors and from OpenCL to AMD graphics processors, and found bugs that are **simpler than or the same as** those found manually by Lahav et al. (2016) and by Wickerson et al. (2015).
- We used our technique to aid the development of a **refined MCM for Nvidia graphics processors** that supports an efficient mapping from OpenCL.

1. M. Batty, A. F. Donaldson, and J. Wickerson, *Overhauling SC Atomics in C11 and OpenCL*, in POPL 2016.
2. K. Nienhuis, K. Memarian, and P. Sewell, *An Operational Semantics for C/C++11 Concurrency*, in OOPSLA 2016.
3. O. Lahav, N. Giannarakis, and V. Vafeiadis, *Taming Release-Acquire Consistency*, in POPL 2017.
4. V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Zappa Nardelli, *Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it*, in POPL 2015.
5. O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer, *Repairing Sequential Consistency in C/C++11*. Draft, 2016.
6. J. Wickerson, M. Batty, B. M. Beckmann, and A. F. Donaldson, *Remote-Scope Promotion: Clarified, Rectified, and Verified*, in OOPSLA 2015.

We gratefully acknowledge the financial support of a Research Fellowship from the Royal Academy of Engineering and the Lloyd's Register Foundation (Batty), a Research Chair from the Royal Academy of Engineering and Imagination Technologies (Constantinides), an EPSRC Impact Acceleration Award, EPSRC grants EP/I020357/1, EP/K034448/1, and EP/K015168/1, and a gift from Intel Corporation.