Formal Reasoning about the C11 Weak Memory Model

Viktor Vafeiadis

Max Planck Institute for Software Systems (MPI-SWS)

viktor@mpi-sws.org

Abstract

This abstract introduces the C11 weak memory model, summarises known verification results, and discusses some open problems.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Concurrency; Weak Memory Models; Program Logic

1. A Primer on Weak Memory Models

The memory model is what defines the semantics of memory accesses in multithreaded programs. The simplest memory model is *sequential consistency* (SC), which interleaves the memory accesses of each thread to construct a total order among them and requires that memory loads return the value written by the last store to the same address preceding them in this total order.

Sequential consistency might be a nice and simple way of thinking about concurrency, but does not reflect current practice. As an example, consider the following two-threaded program:

$$\begin{array}{l} x := 1; \\ a := y; \end{array} \left\| \begin{array}{l} y := 1; \\ b := x; \end{array} \right. \qquad \text{(SB)}$$

where initially $x = y = 0$. According to SC, the program can never terminate with $a = b = 0$ because at least one of the stores to x and y has to complete before the loads of y and x. All recent hardware architectures (e.g., x86, PowerPC, ARM, Itanium), however, do allow this outcome, and indeed $a = b = 0$ has been observed by testing real hardware implementations. At the hardware level, this is typically explained in terms of store buffering. For example, in the x86-TSO model [9], the stores to x and y are put into the corresponding processor's store buffer and may not be propagated to the other processors until after the loads have taken place.

Store buffering alone, however, is not sufficient to explain all observable weak memory behaviours. As an illustration, consider the independent reads of independent writes (IRIW) program below, where initially $x = y = 0$.

$$x := 1; \left\| \begin{array}{l} c := x; \\ a := y; \end{array} \right\| \begin{array}{l} d := y; \\ b := x; \end{array} \left\| \; y := 1; \right. \qquad \text{(IRIW)}$$

On Power and ARM (but not on x86-TSO), this program may return $c = d = 1$ and $a = b = 0$, indicating that the two middle threads observed the (independent) stores to x and y happen in a different order.

CPP '15, January 13–14, 2015, Mumbai, India.
Copyright is held by the owner/author(s).
ACM 978-1-4503-3300-9/15/01.
http://dx.doi.org/10.1145/2676724.2693181

2. What is the C11 Memory Model?

The C11 memory model was introduced by the C++ 2011 standard [6] based on the work of Boehm and Adve [3], and defines the semantics of concurrent C/C++ programs.

In order to allow maximum flexibility, C11 provides a spectrum of memory accesses, each providing different synchronisation guarantees and having different implementation costs. These range from cheap normal (non-atomic) accesses to expensive SC-atomic accesses. On the one end of the spectrum, races on non-atomic accesses result in completely undefined behaviour (they are treated as programming errors); on the other end, SC-atomic accesses are globally synchronised, and so if races are confined to SC-accesses, then the program behaves as though it were running under interleaving semantics.[1]

Between the two extremes, C11 provides acquire atomic loads, release atomic stores, relaxed atomic accesses, and consume loads, that have weaker synchronisation guarantees. In particular, they allow the non-SC behaviours of the (SB) and (IRIW) examples.

What makes understanding C11 much more challenging than hardware memory models such as TSO is not only the many kinds of accesses it provides, but also the way in which it is defined. In C11, the meaning of a program is defined to be a set of consistent executions. An execution can be though of as a generalised program trace: it is a graph with vertices corresponding to the program's memory accesses and with edges corresponding to the various orderings guarantees provided by the memory model. These include the program order, the reads-from relation, each variable's modification order, a total order on SC-accesses, and others. An execution is called consistent if it satisfies a bunch of constraints (a.k.a., axioms) specified by the model. A typical such constraint might say that reads should read from some non-later write to the same address that wrote the same value as the one returned by the read. A formal presentation of the model can be found in Batty et al. [1].

3. Verification Results: Verifying C11 Programs

The constraint-based definition of the C11 model is very global and, while it may be used to determine whether a given program outcome is allowed, it does not directly lead to any techniques for reasoning about program correctness.

Thus, our first objective was to develop such techniques. One basic property we are aiming for is compositionality: the ability to decompose a reasoning about a program to reasoning about its parts. Another is that we want to reason in terms of the source code, and not by considering all possible executions filtered by those satisfying the model's constraints.

In essence, we want to develop a usable axiomatic semantics for verifying concurrent C11 programs. In our work, we have focused on the release-acquire fragment of C11, and have developed two program logics for reasoning about programs in this fragment.

- *Relaxed Separation Logic (RSL)* [14] is a fairly simple logic that adapts the notion of ownership found in concurrent separation

[1] This is known as the DRF-SC theorem: it holds for a fragment of C11 [3].

logic [8] to the release-acquire fragment of C11. The main idea is to use separation logic's rules to ensure the absence of data races on normal memory accesses, and to introduce special rules for atomic accesses. An important feature of RSL is that it allows ownership transfer via release-acquire accesses.

- *GPS* [13] is a more advanced logic that incorporates the crucial features found in modern concurrent program logics, namely ghost assertions, protocols, and separation. GPS goes much beyond what can be achieved in RSL, and can be used to verify advanced weak-memory algorithms operating under release-acquire semantics. As an example, we have recently applied GPS to verify a weak memory implementation of user-mode RCU [5], a synchronisation primitive used in Linux.

4. Compilation Results: Verifying C11 Compilers

To ensure that any specifications proved about programs running over the C11 memory model actually hold when these programs are run, we also need to verify that the compilers that translate C/C++ down to machine code are faithful to the C11 memory model.

A first set of results in this area was by Batty et al. [1, 2, 10], who verified the proposed compilations of the C11 atomic primitive memory accesses to the corresponding code snippets in x86-TSO, PowerPC, and ARM. These results are very useful, but assume that the remainder of the compiler preserves the exact pattern of memory accesses, an assumption that is rarely true in any realistic scenario.

To deal with compiler transformations that change the memory access patterns, Sevcik [11] developed a formal model in which he proved the correctness of a collection of abstract transformations on programs without data races. Sevcik's development did not consider the C11 model directly, but rather a trace-based one intended to be equivalent to the SC+NA fragment of the C11 model. Later, Morisset et al. [7] adapted this result to actual C11 setting, thereby handling transformations on non-atomic memory accesses in C11.

More recently, we also considered transformations on atomic memory accesses [15]. Much to our surprise, we showed that many of these transformations are actually invalid as C11 source-to-source transformations. This was largely due to errors in the formal definition of the C11 memory model. We then considered various possible fixes to the errors of the C11 model, and showed that the most of the transformations intended to be correct are indeed correct in the rectified models.

Following the work on determining which program transformations are sound, Soham Chakraborty and I are looking at validating LLVM compiler optimisations with respect to what is allowed by the C11 memory model.

5. Conclusion: Some Open Problems

In the last few years, a lot of progress has been made in understanding and reasoning about weak memory models including C11. Nevertheless, many difficult problems remain to be solved. At the semantic level, we need to come up with a decent definition for C11-style relaxed atomic accesses that do not fall foul of the 'out of thin air' read problem (cf. [14, 15]). We also need to determine a useful semantics for strong memory fences in C11, as the current semantics is too weak.

From the verification perspective, there are many interesting problems to be considered. One should develop proof rules for reasoning about other features of the C11 model, such as memory fences, and consume atomic loads. A more challenging problem might be to develop suitable notions of atomicity and refinement under weak consistency, or to state and verify liveness properties in the presence of weak memory consistency. Another direction would be to develop a verified compiler for C11 in an analogous way to Sevcik et al. [12] or to verify some important compiler optimisations with respect to concurrency.

In the remainder, I will briefly discuss one other verification problem, that of checking robustness. We call a program running under a memory model X *robust* against a different memory model Y, if the program has identical behaviours under X and Y.

While there exist good robustness checking procedures for hardware memory models against SC (e.g., [4]), it is still unknown whether similar results could be derived for the release-acquire fragment of the C11 memory model against SC or TSO.

The precise relationship between release-acquire and TSO is also not fully understood. What is known is that release-acquire is strictly weaker than TSO. The smallest known examples that can distinguish release-acquire from TSO are the IRIW program with two variables and four threads, and the following program with two variables and only two threads but with store-store races.

$$\begin{array}{r||l} y := 2; & x := 2; \\ x := 1; & y := 1; \\ a := x; & \\ b := y; & \end{array} \tag{2+2W}$$

The outcome $a = b = 2$ is not possible under TSO, but it is allowed under C11 release-acquire semantics.

One may therefore conjecture that if the program contains up to three threads and no store-store races, then its release-acquire and TSO behaviours coincide. How to prove this conjecture, however, is not clear.

References

[1] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL*, pages 55–66, 2011.

[2] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: From C++11 to POWER. In *POPL*, pages 509–520, 2012.

[3] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *PLDI*, pages 68–78, 2008.

[4] A. Bouajjani, R. Meyer, and E. Möhlmann. Deciding robustness against total store ordering. In *ICALP (2)*, pages 428–440, 2011.

[5] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole. User-level implementations of read-copy update. *IEEE Trans. Parallel Distrib. Syst.*, 23(2):375–382, 2012.

[6] ISO/IEC 14882:2011. Programming language C++, 2011.

[7] R. Morisset, P. Pawan, and F. Zappa Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *PLDI*, pages 187–196, 2013.

[8] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[9] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *TPHOLs*, pages 391–407, 2009.

[10] S. Sarkar, K. Memarian, S. Owens, M. Batty, P. Sewell, L. Maranget, J. Alglave, and D. Williams. Synchronising C/C++ and POWER. In *PLDI*, pages 311–322, 2012.

[11] J. Sevcik. Safe optimisations for shared-memory concurrent programs. In *PLDI*, pages 306–316, 2011.

[12] J. Sevcik, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22, 2013.

[13] A. Turon, V. Vafeiadis, and D. Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*, pages 691–707, 2014.

[14] V. Vafeiadis and C. Narayan. Relaxed separation logic: A program logic for C11 concurrency. In *OOPSLA*, pages 867–884, 2013.

[15] 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.