Robustness against the C/C++11 Memory Model

Roy Margalit
Tel Aviv University
Tel Aviv, Israel
roy.margalit@cs.tau.ac.il

Abstract

Concurrency is extremely useful, however reasoning about the correctness of concurrent programs is more difficult than ever. Sequential consistency (SC) semantics provide the ability to reason about correctness, however these come at a high synchronization cost. C11 introduced a memory model that was supposed to help with these problems, attempting to provide balance between performance and reasoning. However, this model was much more complicated than expected, proving to be a challenge even for domain experts. We propose a method that enables the programmer to reason about correctness with SC semantics without compromising performance. By proving robustness of a program it can only exhibit SC behaviors and can thus be reasoned about with SC semantics.

CCS Concepts

• **Theory of computation** → *Verification by model checking*; *Concurrent algorithms*; *Program semantics*; *Program verification*; *Program analysis*; • **Software and its engineering** → *Software verification.*

Keywords

weak memory models, C/C++11, robustness, shared-memory concurrency, release/acquire

ACM Reference Format:
Roy Margalit. 2024. Robustness against the C/C++11 Memory Model. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3650212.3685549

1 Problem Description

Many concurrent algorithms assume sequential consistency (SC) (i.e., executing the program by interleaving statements from the various threads at some order over a shared memory with no caching). In practice, modern programming languages do not guarantee SC because they are compiled to hardware which performs various optimizations such as out-of-order execution and separate caches for each CPU core. To make things worse, each hardware has its own memory model (PowerPC, SPARC, x86, ARM, Itanium, etc.).

Because of this, the programmer has no easy way to reason about the correctness of their program. This problem is especially difficult for programs that are required to be correct over multiple architectures.

Robustness. Programming concurrent problems under memory models weaker than SC is notoriously hard and error-prone. Reasoning about the correctness of these problems even more so. When programmers manage to avoid data races, the DRF (data-race-freedom) guarantee allows them to safely imagine a heaven of SC multiprocessors and SC-preserving compilers [4]. For a variety of programs, however, avoiding races altogether is too restrictive or costly. Interestingly, often this does not mean that the program may behave under a weak memory model differently than it would behave under SC. Accordingly, there is a need to develop *robustness* criteria (i.e., conditions that ensure that a given program has only SC behaviors), which are more precise then DRF, and accompany them with robustness verification methods and tools. Safety verification can be then reduced to robustness verification plus safety verification assuming SC [17].

Thus, if we are able to verify that a program is robust against a memory model and that the program is correct assuming SC, we can deduce that the program is correct under that memory model.

C11 and C++11. The C11 [11] memory model was introduced with the aim of reducing these problems. Instead of the programmer having to write platform specific code for each architecture, it provides a uniform memory model. The compiler will use a different compilation scheme based on the hardware target, mapping the memory accesses in C11 to the appropriate assembly instructions [50] while preserving a subset of behaviors from the C11 memory model. Thus, as long as the compilation scheme is correct, the programmer may execute their program on different hardware featuring different memory models without having to reason about each specific hardware. The C11 memory model attempts to balance between the ability of the programmer to reason about their program and the need for efficient machine code.

Prior work. Robustness against weak memory semantics was studied before for *hardware* models, especially in the context of automatically enforcing robustness by inserting memory fences and other synchronization primitives (see, e.g., [6, 16, 19, 20], as well as [5] for a practical approximate generic approach).

In particular, robustness against TSO (and its PSO variant) [7, 28, 47] received considerable attention, e.g., [2, 3, 13–15, 17, 18, 26, 38–40, 46]. In addition, verifying (trace based) robustness against TSO was shown in [15] to be PSPACE complete.

Less work was devoted to robustness against a *programming language* concurrency semantics. The well-known DRF guarantee [4, 25] is a simple robustness criterion, e.g., for a strengthened version of C11 [10, 33], but it is too weak, as (low-level) synchronizations naturally involve data-races, and often do not imply non-robustness. [44] proposed an (approximate and incomplete) method

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3685549

that uses CDSchecker [45] for restricting non-SC behaviors of C11 programs. For a particular class of "server client programs", it was shown in [31] that certain simple fence insertion strategies ensures robustness.

Other works targeting C11 [24, 36, 37, 41, 49] attempted to find races between non-atomics which cause undefined behavior. However in these works, Only the current execution is tested where the scheduler is either controlled by the operating system [36, 49] or the testing framework tries to introduce the races by controlling the scheduler [24, 37, 41]. Still none of these approaches can model the entirety of the C11 Memory Model and cannot identify all behaviors.

On the other spectrum, Stateless model checkers exhaustively scan all the executions of the program. While super precise, the memory consumption is much higher as they store the entire execution graph in memory [29, 30].

2 Goal Statement

The goal of this research is to provide the programmer with tools to better reason about the correctness and robustness of their C11 progams. Reasoning about the C11 memory model is hard even for domain experts. Even implementing a well known algorithm, such as Peterson's lock, is not simple [52, 53]. Expert programmers experienced with weak memory models are still prone to errors. Even informal [53] and formal [22] proofs can incorrectly prove safety of programs. This may result in programmers refraining from using relaxed accesses altogether.

We would like to avoid this problem by providing an algorithm which can check whether a program is robust against the C11 memory model. If a program is proved to be robust, we only require to verify its correctness under SC, a much easier feat than proving correctness under C11. The final product is the creation of a tool which will automatically verify this.

3 Method

The main challenge is developing an algorithm for proving robustness. The correctness of this algorithm will be proved using a formal proof assistant (Coq). Once such an algorithm is created, we will implement it. The implementation is then verified by comparing manually verified robustness examples against the automated result, thus showing effectiveness of our tool. The implementation will be verified by comparing manually verified robustness examples against the automated result. Further more, for unknown examples a robustness violation trace will be extracted and manually verified to be a proof on non robustness.

Static vs. dynamic. We would like to evaluate our approach in two contexts, static and dynamic. A static context where all possible executions of the program are checked for robustness. This approach guarantees that no robustness violations can be missed as all executions are examined. The disadvantage is that this approach does not scale beyond small programs. A dynamic context, on the other hand, allows us to execute the program in-the-wild with proper instrumentation. For each such an execution we follow a single possible trace of the execution which we can analyze to identify if it potentially predicts a robustness violation. This can be done in both on-the-fly manner, where we analyze the execution as we run the program, and post-mortem manner where the trace of all memory accesses from the execution is recorded and later analyzed.

We will evaluate the two approaches to verify that they produce the same robustness claim for programs the static approach can handle. Thus, if the dynamic verification reports non robustness we expect the static one to report it as well.

Benchmarks. We will make use of existing benchmarks from Trencher [13], a robustness checker for x86-TSO. The benchmarks will include concurrent data structures, mutual exclusions protocols, as well as other common synchronization idioms adapted to C11.

Performance critical widgets that are commonly used in user code. Proving their robustness will affect the robustness of the entire program (if they are correctly placed behind SC fences). One such example is Chase-Lev dequeue which is a commonly used work-stealing queue that is hard to implement correctly using weak memory [35].

We will test dynamic robustness verification by running it on large programs to see how well it scales. For testing large programs we will use both synthetically inflated programs and real world programs.

4 Preliminary Work

We prove that (assuming bounded data domain) execution-graph robustness against RC20 (a subset of a corrected version of C11) is PSPACE-complete (the same as verification under SC and robustness against x86-TSO) We develop a tool for verifying execution-graph robustness against RC20 and use it to evaluate several challenging synchronization algorithms.

In this section we describe our preliminary results. We wrote two papers about verifying robustness of C11 memory model subsets with static analysis.

Robustness Against Release/Acquire. Release/acquire (RA), the fragment of the C11 memory model [11] consisting of release stores, acquire loads and acquire-release read-modify-writes (RMWs), is a particularly useful and well-behaved weak memory model [31]. It is weaker than sequential consistency (SC) [34] and allows higher performance implementations. For example, x86-TSO [47] provides RA "for free" (its memory model is stronger than RA), and PowerPC [42] implements RA using 'lightweight sync' instructions rather than more expensive 'full sync' instructions which are needed for SC.

We developed a decision procedure that checks whether a given concurrent program is robust against RA. To achieve this, we showed how this verification problem can be reduced to a state reachability problem under a (finite state) instrumented SC memory. Roughly speaking, this memory keeps track of the relevant parts of the generated execution graph and uses this information for monitoring that RA execution graphs cannot diverge from SC ones. We proved that our approach is sound and precise. In particular, it follows that this verification problem for programs with bounded data domain is PSPACE-complete.

Our approach is straightforwardly extended to handle C11's *non-atomic accesses*. A data-race on a non-atomic access is considered an undefined behavior, and, thus, robustness of a program should also imply that it has no data-races on non-atomic accesses. Since robust

programs have only SC executions, checking for data-races can be done using standard techniques. For completeness, we incorporated these checks in our method simultaneously to the verification of robustness against RA.

We have implemented our method in a prototype tool, called *Rocker*, using Spin [27] as a back-end model checker under SC. We used *Rocker* to verify the robustness of several concurrent algorithms, including Peterson's mutual execution adaptations for RA [53], sequence locks [12] and user-mode read-copy-update (RCU) implementations [21]. In particular, we observed that robustness under RA is a useful property, allowing one, in many cases, to think in terms of SC while running on a weaker model.

Examples of verified programs and the time taken to verify are available in Table 1. *Res* shows robustness against RA, *SC* is a comparison to verification without instrumentation, and *Trencher* is robustness checker tool for TSO. This work was published and is available at [32].

Table 1: Robustness against RA verified with Rocker

Program	Res	#T	LoC	Time		SC	Trencher TSO	
							Res	Time
dekker-sc	✗	2	43	4.2	(100%)	1.3	✗	5.9
dekker-tso	✓	2	49	5.2	(100%)	1.3	✓	5.9
peterson-ra	✓	2	44	5.8	(100%)	1.2	✓	5.8
peterson-ra-dmitriy	✓	2	36	4.3	(100%)	1.2	✓	5.5
peterson-ra-bratosz	✗	2	28	3.4	(100%)	1.1	✗	5.6
lamport2-tso	✗	2	69	13.7	(100%)	1.3	✓	8.2
lamport2-ra	✓	2	79	18.9	(99%)	1.4	✓	7.8
spinlock4	✓	4	66	6.4	(80%)	1.6	✓	6.8
ticketlock4	✓	4	49	22.6	(25%)	7.5	✓	23.4
seqlock	✓	4	49	20.7	(16%)	3.4	✓	8.9
rcu	✓	4	74	67.6	(10%)	2.2	✗*	-
rcu-offline	✓	3	215	137.9	(50%)	18.3	✗*	-
chase-lev-tso	✗	3	57	4.9	(100%)	31.3	✓	128.1
chase-lev-ra	✓	3	61	67.1	(8%)	38.1	✓	108.3

Robustness Against RC20. We extended our results to a much larger fragment of C11, which also includes relaxed reads/writes/RMWs and release/acquire fences. We showed that robustness against this model is reduced to a reachability problem under an instrumented SC semantics.[1] Roughly speaking, the instrumentation maintains a finite collection of (rather intricate) properties of the generated execution history (which can be unbounded for programs with loops) that are used to monitor for states where threads may observe (either by reading or by overwriting) values under the weak semantics that cannot be observed under SC. Based on this reduction, we obtained a decision procedure for robustness, which we have also implemented and experimented with (using Spin [27] for the reachability analysis under SC). In particular, it follows that robustness verification for programs with bounded data domain is PSPACE-complete.

[1]Equivalently, as we do in our implementation, one can think about reachability under SC of an instrumented program.

For similar reasons why programming is more difficult with relaxed accesses, robustness verification for relaxed accesses is also more challenging. Indeed, the instrumented semantics requires much finer distinctions maintaining multiple views for each thread allowing us to compare the observable values under SC to those observable in the relaxed semantics. While the instrumentation is heavier than the one in [32], our experiments show that this has only a minor effect on the practical verification times.

While handling relaxed accesses, we have identified a drawback of all (to the best of our knowledge) existing robustness criteria, which becomes critical for robustness against a semantics that includes relaxed accesses. It is related to the question of what exactly constitutes a program behavior (and, in turn, what is a robustness violation?). The simplest answer identifies program behaviors with sets of reachable program states, where program states consist of the values of the different program counters and local variables. However, [19] observed that, while desirable, a state-based notion of robustness requires a full-fledged verification under the weak semantics. The latter can be extremely difficult from a computational complexity point of view (non-primitive recursive for TSO [8, 9] and undecidable for RA [1]). Thus, researchers have resorted to stronger robustness definitions, in which program behaviors are taken to be sets of "annotated" histories of the executions of the programs (called *traces* or *execution graphs*), and robustness requires that all such histories generated by the program under the weak semantics can be also generated under SC.

However, we observed here that both state-based and history-based robustness notions do not allow the benign programming idiom of *speculative* reads. Roughly speaking, weak accesses can be efficiently used to speculatively read some shared data, requiring a validation mechanism (naturally implemented with stronger accesses) before the read values are being used. If validation fails, the speculations are disposed (and often retried). Observing stale speculative values under relaxed semantics (which cannot be observed under SC) may be perfectly fine, but it does imply a reachable intermediate program state that is not reachable under SC, resulting in a robustness violation. Nevertheless, if the validation mechanism can be proved to throw away stale speculative values before they are actually being used, this optimization should be considered as a benign temporary robustness violation that can never harm the program's safety. Since speculative reads are a main use of relaxed accesses [51], and using them correctly has shown to be rather challenging [12]. For the model that we studied here, this issue becomes more acute.

Observational Robustness: To address this drawback, we proposed a novel notion of robustness, which we called *observational robustness*. It is a history-based notion (to make is computationally attainable), but it allows the program to have some non-SC histories provided that they can be "fixed" to SC histories without affecting any value that the program used. To do so, we included a dependency relation (akin to the one included in modern hardware models, see, e.g., [23, 48]) in the program's execution graphs and allow reads to read stale values (violating SC) as long as no further operation depends on these reads.

As we did for usual robustness, we showed that observational robustness can be verified by inspecting *only* SC executions of the program, which, again, can be reduced to reachability in an

Table 2: Robustness against C11 verified with Rocker

Program	#T	LoC	Rob		Time (sec)					
			R	O	R		O		SC	
arc	3	102	✓	✓	67.1	(8%)	71.2	(8%)	3.9	(26%)
peterson	2	37	✗	✗	1.1	(100%)	1.1	(100%)	0.9	(100%)
peterson-fix1	2	37	✓	✓	1.1	(100%)	1.1	(100%)	0.9	(100%)
peterson-fix2	2	39	✓	✓	1.1	(100%)	1.1	(100%)	0.9	(100%)
singleton	4	139	✓	✓	2.8	(96%)	3.0	(97%)	1.1	(100%)
singleton-rlx	4	147	✓	✓	2.8	(96%)	3.1	(97%)	1.1	(100%)
dekker-rlx	2	61	✓	✓	1.5	(100%)	1.6	(100%)	0.9	(100%)
seqlock rdmw	3	84	✗	✓	7.1	(100%)	59.7	(12%)	30.9	(4%)
seqlock fence	3	90	✗	✓	8.0	(100%)	81.8	(10%)	39.9	(3%)
seqlock rdmw-rw	3	81	✗	✓	6.4	(100%)	53.5	(13%)	31.9	(3%)
seqlock fence-rw	3	87	✗	✓	6.7	(100%)	74.6	(9%)	41.5	(3%)

instrumented SC semantics. These results required certain technical novelties as we have to carefully generate an SC execution from any observational robustness violation (that could have diverged from SC early on), as well as to devise a specialized taint tracking mechanism for identifying dependencies on possibly stale values. We note that our observational robustness verification is sound but, unlike our result for usual robustness, imprecise. Still, we showed that it verifies the challenging examples of [12] deeming them as robust and safe.

Examples of verified programs and the time taken to verify are available in Table 2. *R* stands for robustness, *O* stands for observational robustness and *SC* is a comparison to verification without instrumentation. This work was published and is available at [43].

Dynamic checking. Using the above method, we develop the first dynamic verification method for program robustness. This means that we consider one trace at a time, and check that the atomics in that trace are used in a way that cannot generate weak behaviors. To do so, like dynamic race detectors, we instrument the program with code for identifying non-robustness witnesses (our analog of data races). The instrumentation we develop is based on our previous approach but it is based on *vector clocks* that are more compact and more efficiently maintained during the analysis, compared the instrumentation used for static verification of robustness in [32, 43]. We achieve soundness and completeness, and, importantly, the memory consumption for the analysis of the trace is polynomial in the number of threads and variables, and does not depend on the length of the trace. We have implemented our approach in a prototype post-mortem robustness analyzer, and evaluated it on multiple examples. All in all, combined with TSan [49] (or any other tool assuming SC semantics) for detecting races, our results show a viable option for dynamic monitoring of C11 programs.

References

[1] Parosh Aziz Abdulla, Jatin Arora, Mohamed Faouzi Atig, and Shankaranarayanan Krishna. 2019. Verification of programs under the release-acquire semantics. In *PLDI*. ACM, New York, NY, USA, 1117–1132. https://doi.org/10.1145/3314221.3314649

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Magnus Lång, and Tuan Phong Ngo. 2015. Precise and sound automatic fence insertion procedure under PSO. In *NETYS*. Springer International Publishing, Cham, 32–47.

[3] Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Tuan-Phong Ngo. 2015. The best of both worlds: Trading efficiency and optimality in fence insertion for TSO. In *ESOP*. Springer-Verlag New York, Inc., New York, 308–332. https://doi.org/10.1007/978-3-662-46669-8_13

[4] Sarita V. Adve and Mark D. Hill. 1990. Weak ordering—a new definition. In *ISCA* (Seattle, Washington, USA). ACM, New York, 2–14. https://doi.org/10.1145/325164.325100

[5] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. 2017. Don't sit on the fence: a static analysis approach to automatic fence insertion. *ACM Trans. Program. Lang. Syst.* 39, 2, Article 6 (May 2017), 38 pages. https://doi.org/10.1145/2994593

[6] Jade Alglave and Luc Maranget. 2011. Stability in weak memory models. In *CAV* (Snowbird, UT). Springer-Verlag, Berlin, Heidelberg, 50–66. http://dl.acm.org/citation.cfm?id=2032305.2032311

[7] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding cats: modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.* 36, 2, Article 7 (July 2014), 74 pages. https://doi.org/10.1145/2627752

[8] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. 2010. On the verification problem for weak memory models. In *POPL* (Madrid, Spain). ACM, New York, 7–18. https://doi.org/10.1145/1706299.1706303

[9] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. 2012. What's decidable about weak memory models?. In *ESOP* (Tallinn, Estonia). Springer-Verlag, Berlin, Heidelberg, 26–46. https://doi.org/10.1007/978-3-642-28869-2_2

[10] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The problem of programming language concurrency semantics. In *ESOP*. Springer, Berlin, Heidelberg, 283–307. https://doi.org/10.1007/978-3-662-46669-8_12

[11] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *POPL*. ACM, New York, 55–66. https://doi.org/10.1145/1925844.1926394

[12] Hans-J. Boehm. 2012. Can Seqlocks get along with programming language memory models?. In *MSPC* (Beijing, China). ACM, New York, 12–20. https://doi.org/10.1145/2247684.2247688

[13] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. 2013. Checking and enforcing robustness against TSO. In *ESOP* (Rome, Italy). Springer-Verlag, Berlin, Heidelberg, 533–553. https://doi.org/10.1007/978-3-642-37036-6_29

[14] Ahmed Bouajjani, Constantin Enea, Suha Orhun Mutluergil, and Serdar Tasiran. 2018. Reasoning about TSO programs using reduction and abstraction. In *CAV*. Springer, Cham, 336–353.

[15] Ahmed Bouajjani, Roland Meyer, and Eike Möhlmann. 2011. Deciding robustness against total store ordering. In *ICALP*. Springer, Berlin, Heidelberg, 428–440.

[16] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. 2007. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *PLDI* (San Diego, California, USA). ACM, New York, 12–21. https://doi.org/10.1145/1250734.1250737

[17] Sebastian Burckhardt and Madanlal Musuvathi. 2008. Effective program verification for relaxed memory models. In *CAV* (Princeton, NJ, USA). Springer-Verlag, Berlin, Heidelberg, 107–120. https://doi.org/10.1007/978-3-540-70545-1_12

[18] Jabob Burnim, Koushik Sen, and Christos Stergiou. 2011. Sound and complete monitoring of sequential consistency for relaxed memory models. In *TACAS*. Springer, Berlin, Heidelberg, 11–25.

[19] Egor Derevenetc. 2015. *Robustness against relaxed memory models*. Ph. D. Dissertation. University of Kaiserslautern. http://kluedo.ub.uni-kl.de/frontdoor/index/

index/docId/4074
[20] Egor Derevenetc and Roland Meyer. 2014. Robustness against Power is PSpace-complete. In *ICALP*. Springer, Berlin, Heidelberg, 158–170.
[21] Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais, and Jonathan Walpole. 2012. User-level implementations of read-copy update. *IEEE Trans. Parallel Distrib. Syst.* 23, 2 (Feb. 2012), 375–382. https://doi.org/10.1109/TPDS.2011.159
[22] Simon Doherty, Brijesh Dongol, Heike Wehrheim, and John Derrick. 2019. Verifying C11 programs operationally. In *PPoPP*. ACM, New York, NY, USA, 355–365. https://doi.org/10.1145/3293883.3295702
[23] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA. In *POPL 2016* (St. Petersburg, FL, USA). ACM, New York, 608–621. https://doi.org/10.1145/2837614.2837615
[24] Mingyu Gao, Soham Chakraborty, and Burcu Kulahcioglu Ozkan. 2023. Probabilistic Concurrency Testing for Weak Memory Programs. In *ASPLOS*. ACM, New York, NY, USA, 603–616. https://doi.org/10.1145/3575693.3575729
[25] Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill. 1992. Programming for different memory consistency models. *J. Parallel and Distrib. Comput.* 15, 4 (1992), 399 – 407. https://doi.org/10.1016/0743-7315(92)90052-O
[26] Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. 2012. Show no weakness: sequentially consistent specifications of TSO libraries. In *DISC* (Salvador, Brazil). Springer-Verlag, Berlin, Heidelberg, 31–45. https://doi.org/10.1007/978-3-642-33651-5_3
[27] Gerard J. Holzmann. 1997. The model checker SPIN. *IEEE Transactions on software engineering* 23, 5 (1997), 279–295.
[28] SPARC International Inc. 1994. *The SPARC architecture manual (version 9)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
[29] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2017. Effective stateless model checking for C/C++ concurrency. *Proc. ACM Program. Lang.* 2, POPL, Article 17 (Dec. 2017), 32 pages. https://doi.org/10.1145/3158105
[30] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Model checking for weakly consistent libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 96–110. https://doi.org/10.1145/3314221.3314609
[31] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming release-acquire consistency. In *POPL* (St. Petersburg, FL, USA). ACM, New York, 649–662. https://doi.org/10.1145/2837614.2837643
[32] Ori Lahav and Roy Margalit. 2019. Robustness against release/acquire semantics. In *PLDI*. ACM, New York, NY, USA, 126–141. https://doi.org/10.1145/3314221.3314604
[33] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *PLDI*. ACM, New York, 618–632. https://doi.org/10.1145/3062341.3062352
[34] Leslie Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers* 28, 9 (1979), 690–691.
[35] Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. 2013. Correct and Efficient Work-stealing for Weak Memory Models. In *PPoPP 2013* (Shenzhen, China). ACM, 69–80. https://doi.org/10.1145/2442516.2442524
[36] Christopher Lidbury and Alastair F. Donaldson. 2017. Dynamic race detection for C++11. In *POPL*. ACM, New York, NY, USA, 443–457. https://doi.org/10.1145/3009837.3009857
[37] Christopher Lidbury and Alastair F. Donaldson. 2019. Sparse record and replay with controlled scheduling. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 576–593. https://doi.org/10.1145/3314221.3314635
[38] Alexander Linden and Pierre Wolper. 2011. A verification-based approach to memory fence insertion in relaxed memory systems. In *SPIN* (Snowbird, UT). Springer-Verlag, Berlin, Heidelberg, 144–160. http://dl.acm.org/citation.cfm?id=2032692.2032707
[39] Alexander Linden and Pierre Wolper. 2013. A verification-based approach to memory fence insertion in PSO memory systems. In *TACAS* (Rome, Italy). Springer-Verlag, Berlin, Heidelberg, 339–353. https://doi.org/10.1007/978-3-642-36742-7_24
[40] Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin Vechev, and Eran Yahav. 2012. Dynamic synthesis for relaxed memory models. In *PLDI* (Beijing, China). ACM, New York, 429–440. https://doi.org/10.1145/2254064.2254115
[41] Weiyu Luo and Brian Demsky. 2021. C11Tester: a race detector for C/C++ atomics. In *ASPLOS*. ACM, New York, NY, USA, 630–646. https://doi.org/10.1145/3445814.3446711
[42] Luc Maranget, Susmit Sarkar, and Peter Sewell. 2012. A tutorial introduction to the ARM and POWER relaxed memory models. http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf.
[43] Roy Margalit and Ori Lahav. 2020. Robustness against release/acquire semantics. In *Submitted to Principles of Programming Languages POPL 2021*.
[44] Yuri Meshman, Noam Rinetzky, and Eran Yahav. 2015. Pattern-based synthesis of synchronization for the C++ memory model. In *FMCAD* (Austin, Texas). FMCAD Inc, Austin, TX, 120–127. http://dl.acm.org/citation.cfm?id=2893529.2893552
[45] Brian Norris and Brian Demsky. 2013. CDSchecker: checking concurrent data structures written with C/C++ atomics. In *OOPSLA* (Indianapolis, Indiana, USA). ACM, New York, 131–150. https://doi.org/10.1145/2509136.2509514
[46] Scott Owens. 2010. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *ECOOP*. Springer-Verlag, Berlin, Heidelberg, 478–503.
[47] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A better x86 memory model: x86-TSO. In *TPHOLs*. Springer, Heidelberg, 391–407. https://doi.org/10.1007/978-3-642-03359-9_27
[48] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the Gap Between Programming Languages and Hardware Weak Memory Models. *PACMPL* 3, POPL, Article 69 (Jan. 2019), 31 pages. https://doi.org/10.1145/3290382
[49] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitry Vyukov. 2011. *Dynamic Race Detection with LLVM Compiler*. Technical Report. Google.
[50] Jaroslav Sevcik and Peter Sewell. 2016. C/C++11 mappings to processors. Retrieved July 21, 2020 from https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html
[51] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. 2017. Chasing Away RAts: Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 161–174. https://doi.org/10.1145/3079856.3080206
[52] Dmitriy V'jukov. 2008. C++ atomics and memory ordering. Retrieved June 23, 2020 from https://bartoszmilewski.com/2008/12/01/c-atomics-and-memory-ordering/#comment-111
[53] Anthony Williams. 2008. Peterson's lock with C++0x atomics. Retrieved October 26, 2018 from https://www.justsoftwaresolutions.co.uk/threading/petersons_lock_with_C++0x_atomics.html

Received 2024-07-07; accepted 2024-07-22