Full Weak Memory Model Support for the `loom` [4] Model Checker

Jaehwang Jung
jaehwang.jung@kaist.ac.kr
KAIST
Republic of Korea

Abstract

In this project, I will implement the full weak memory model support for `loom`, a model checker for concurrent Rust programs.

Keywords: stateless model checking, weak memory model, concurrency

ACM Reference Format:
Jaehwang Jung. 2020. Full Weak Memory Model Support for the `loom` [4] Model Checker. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Motivation

Utilizing low-level shared-memory concurrency lies at the heart of high-performance computing in the era of multicore processors. However, shared-memory concurrency is frustratingly difficult due to its inherent non-determinism lurking everywhere. The two main sources of non-determinism are thread interleaving and the weak memory model. To tame the combinatorial explosion of number of behaviors caused by interleaving, many synchronization algorithm have been proposed and implemented. To understand the unintuitive behaviors of the weak memory model, the memory model has been formalized [2, 5, 9] and standardized.

However, shared-memory concurrency is still a difficult topic even for the seasoned developers. The main reason is the lack of off-the-shelf tools to extensively check the correctness of concurrent programs. The importance of tools cannot emphasize enough. For example, writing and debugging C/C++ has become much easier thanks to AddressSanitizer, which is highly scalable, compatible and deployable. Unfortunately, current tools for concurrency are not capable of establishing similar level of confidence. Tools like ThreadSanitizer are far from complete because they are not capable of enumerating possible executions, nor do they support the full weak memory synchronization primitives. To improve the quality of life for developers, the tools should have sufficient coverage for both type of non-determinism (interleaving and weak memory) while maintaining scalability, compatibility and deployability.

Among many program verification techniques, *model checking* hits the sweet spot for concurrency as it can achieve reasonable correctness guarantees with relatively small effort compared to simple testing and formal verification. For this purpose, several model checking algorithms [1, 7, 8, 10] have been proposed in the literature. For completeness, they try to enumerate all possible behaviors permitted by the memory model. For scalability, they try to reduce the number of executions to explore by eliminating the executions that lead to the behaviors that have already been observed. Thanks to these properties, some of them were successfully applied to real-world concurrency libraries and discovered some bugs [10].

There are some efforts to bring these advancements to the practice. The focus of this project, `loom` [4] is a model checker for Rust based on the CDSChecker [10], the model checking algorithm for C/C++ memory model [2]. [1] `loom` effectively applies model checking to Rust test cases. Just by configuring some conditional compilation flags (which is neatly manged by `cargo`, the build tool for Rust) and wrapping an existing test with the function `loom::model()` one can transform a test case into a model to be checked exhaustively according to C/C++ memory model. Using `loom`, I was able to build a homework grader for the *CS492: Design and Analysis of Concurrent Programs*[2] course that I work as a TA for.

Unfortunately, the implementation of `loom` is far from complete: it doesn't support `SeqCst` accesses, `Release`

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] While Rust doesn't have a standardized formal memory model yet, it is widely agreed that Rust's shared memory concurrency follows that of C/C++ standards. This is also reflexed in the API of Rust Standard Library.
[2] https://github.com/kaist-cp/cs492-concur

fence and most importantly `SeqCst` fence. Lack of these features is currently blocking its adaptation to Rust ecosystem. For example, there was an effort to integrate `loom` into `crossbeam` [3]'s CI pass.[3] `crossbeam` is a widely used Rust library that supports many critical tools for concurrency such as epoch-based garbage collection and work-stealing deque. The effort has stalled because these algorithms rely on complex synchronization based on `SeqCst` fences, which is not supported by `loom`.

In this project, I propose to fix this situation by implementing the full C/C++ weak memory model for `loom`.

2 Approach

1. Preparation
   - Read CDSChecker paper [10].
   - Read `loom` source code (about 6KLOC including comments and blanks) and understand its architecture.
2. Implementation
   - `SeqCst` fence (highest priority)
   - `Release` fence
   - `SeqCst` accesses
3. Further improvement (if above steps are successful)
   - User-friendly reporting facility to produce instruction interleaving/reordering that lead to a specific execution
   - Apply the improvements from RCMC [7]. It based on RC11 [9], a slightly different version of C/C++ memory model that fixes some problems of `SeqCst` in the current memory model. Adaptation of RC11 led to much more efficient algorithm than CDSChecker.

3 Expected Result

If I successfully implement all the features mentioned above, `loom` can be integrated into many Rust project including but limited to `crossbeam`, *e.g.*

- the Rust implementation of PEBR [6]
- the modified version of `crossbeam`'s epoch-based garbage collector[4], which turned out to be buggy[5]
- more application to CS492 homework grader
- ...

References

[1] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2015. Stateless Model Checking for TSO and PSO. In *Tools and Algorithms for the Construction and Analysis of Systems*, Christel Baier and Cesare Tinelli (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 353–367.

[2] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. *SIGPLAN Not.* 46, 1 (Jan. 2011), 55–66. https://doi.org/10.1145/1925844.1926394

[3] Crossbeam Developers. [n.d.]. Crossbeam. https://github.com/crossbeam-rs/crossbeam

[4] Tokio Developers. [n.d.]. Loom. https://github.com/tokio-rs/loom

[5] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-Memory Concurrency. *SIGPLAN Not.* 52, 1 (Jan. 2017), 175–189. https://doi.org/10.1145/3093333.3009850

[6] Jeehoon Kang and Jaehwang Jung. 2020. A Marriage of Pointer- and Epoch-Based Reclamation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 314–328. https://doi.org/10.1145/3385412.3385978

[7] 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

[8] 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* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 96–110. https://doi.org/10.1145/3314221.3314609

[9] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. *SIGPLAN Not.* 52, 6 (June 2017), 618–632. https://doi.org/10.1145/3140587.3062352

[10] Brian Norris and Brian Demsky. 2013. CDSchecker: Checking Concurrent Data Structures Written with C/C++ Atomics. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 131–150. https://doi.org/10.1145/2509136.2509514

[3] https://github.com/crossbeam-rs/crossbeam/pull/487
[4] https://github.com/crossbeam-rs/crossbeam/pull/416
[5] https://github.com/crossbeam-rs/crossbeam/issues/514