VSYNC: Push-Button Verification and Optimization for Synchronization Primitives on Weak Memory Models (Extended Abstract)

Jonas Oberhauser[1,2], Rafael Lourenco de Lima Chehab[1,2], Diogo Behrens[1,2], Ming Fu[1,2], Antonio Paolillo[1,2], Lilith Oberhauser[1,2], Koustubha Bhat[1,2], Yuzhong Wen[2], Haibo Chen[2,3], Jaeho Kim[1,2], and Viktor Vafeiadis[4]

[1]Huawei Dresden Research Center, [2]Huawei OS Kernel Lab, [3]Shanghai Jiao Tong University, [4]Max Planck Institute for Software Systems

1. Motivation

Modern multicore architectures, such as ARM, Power, and RISC-V, follow *weak memory models* (WMMs) [4, 12, 17, 30], which allow them to execute independent memory operations out of order. WMMs are becoming increasingly pervasive (latest releases from Apple [33], Microsoft [32] and Huawei [20] run on ARM). So, a lot of concurrent software designed for older, fairly strong memory models such as SPARC/x86 TSO [31] needs to be ported to these modern WMMs.

The good news is that most software use only synchronization primitives for inter-thread communication (*e.g.*, spinlock, mutexes, read-write locks); provided synchronization primitives are correct, such software work on WMMs out of the box [10]. The bad news is that the synchronization primitives themselves heavily rely on the order of a few key memory operations, and can break in subtle and non-reproducible ways if these operations happen to be executed out of order. Thus WMMs include so-called *barriers*, which enforce some ordering among memory operations by sacrificing the substantial performance gains of WMMs.

As synchronization primitives often lie on the critical path, unnecessary or overly-constrained barriers in synchronization primitives affect the performance of the complete system. For example, a single unnecessary barrier in the spinlock of Linux reduced the performance of the whole kernel by 4% [3]. For this reason, experts spend time and effort in identifying the key memory operations that need to be executed in order, and optimizing the usage of barriers accordingly [1, 2, 16, 29, 35].

Unfortunately, identifying the necessary order of memory operations is an error-prone task, even for experts. For example, the optimization of the barriers in the Linux qspinlock introduced a bug [29] that remained unfixed for three years [16]. This clearly exemplifies the need for automated solutions to correctly add missing barriers and remove redundant ones.

2. Limitations of the State of the Art

The literature provides two basic approaches for inserting barriers for WMMs: either by static analysis (*e.g.*, as in Musketeer [11]) or by robustness checking [13]. Both insert barriers to enforce sequential consistency. They have two limitations: (1) they cannot maximally relax fences because they lift the program to achieve sequentially consistent semantics at the memory access level, which may be stronger than necessary; and (2) they only support explicit fences, which incur much higher overhead than implicit barriers of atomic operations on ARM [28].

We propose an alternative approach that iteratively inserts/removes barriers in the code and checks the correctness of the mutated code with model checkers. While our approach overcomes the limitations of prior approaches, it is not viable with the current state of the art model checking on WMMs.

The problem is that model checking on WMMs either does not scale or cannot detect liveness violations (hangs). Model checkers for WMMs are of two types:

- Stateful model checkers [4, 7, 14, 19, 21, 27, 36] record complete program states, and do not scale beyond tiny examples. For instance, two recent stateful model checkers for WMMs, Power2SC [7] and `rmem` [4], took more than half an hour and multiple days, respectively, when we ran them even on small synchronization primitives, in terms of number of accesses to shared memory and code size.

- Stateless model checkers [8, 9, 23, 24, 25, 26], on the other hand, do not record program states and thus by design scale better, but cannot detect non-terminating program executions. Unfortunately, without detecting non-terminating program executions, optimization would invariably overly relax the barriers and cause the program to hang on real hardware – we experienced this firsthand when we used the recent stateless model checker GenMC [23].

3. Key Insights

Three key insights allow us to design a novel approach to efficiently optimize barriers of synchronization primitives on WMMs, while producing maximally-relaxed results and ensuring safety and termination with a model checker.

Detecting non-termination. We observe that non-termination in synchronization primitives is exclusively caused by *await loops*, *i.e.*, loops that are side-effect-free except in their last iteration. For programs whose non-terminations are confined to such loops, we show that they can be checked by a finite enumeration of finite executions with a certain property. By applying this insight to stateless model checking (SMC), we make it possible for the first time to automatically detect non-termination on WMMs.

Exploiting monotonicity. The state space of possible barrier combinations is huge (exponential in the size of the program). So, it is hopeless to naively search through it (*e.g.*, with a breadth-first-search). Fortunately, barrier relaxations are monotonic [34]: *relaxing an already incorrect barrier combination can never produce a correct one*. Therefore, we can gradually relax one barrier at a time until no further correct relaxation is possible, achieving linear complexity.

Speculating correctness. Model checking a synchronization primitive with a correct barrier combination requires exploring all executions, whereas the same primitive with an incorrect barrier combination only requires exploring one incorrect execution. The former takes significantly longer than the latter, often two orders of magnitude. By using adaptive timeouts, we can speculate on the correctness of the barrier combination, aborting long runs of the model checker after the timeout, provided we *fully verify the final combination* found in the iterative optimization.

4. Main Artifacts

Our main artifact is the VSYNC framework, which allows one to efficiently optimize the barriers in synchronization primitives on WMMs, producing maximally-relaxed results and ensuring safety and termination. VSYNC consists of two main novel components (see Fig. 1):

- Adaptive linear relaxation (ALR), an efficient barrier optimization algorithm based on adaptive speculation.
- Await model checking (AMC), an extension of SMC that can detect non-terminating await loops on WMMs.

We developed AMC in C++ on top of GenMC [24, 26], a highly advanced SMC from the literature. We implemented ALR and supporting components in Golang.

We ran VSYNC on more than 15 synchronization primitives from both the literature and industry. With AMC we could detect bugs in open source and industry code [18, 22] implemented for WMM by experts. We evaluated our optimized versions against comparable implementations by experts with several microbenchmarks as well as a concurrent DB on high performance ARM servers.

Our secondary artifact is a set of provably-correct high-performance synchronization primitives, which are suitable for practical use in industry.

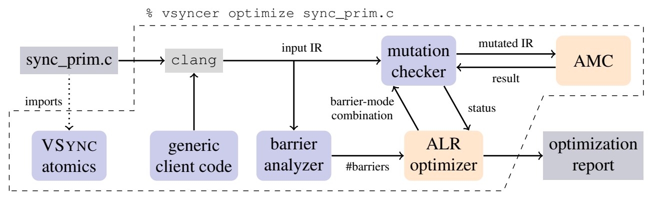

Figure 1: `vsyncer optimize` reports the maximally-relaxed barrier-mode combination of a synchronization primitive.

5. Key Results and Contributions

Our contributions are:

- ALR: a novel algorithm for traversing the exponential search space of barrier optimizations in a linear number of steps, which caps all but the last run of the model checker with an adaptively estimated timeout.
- AMC: a novel method for detecting certain kinds of non-terminating loops on WMMs, which suffices for verifying synchronization primitives.

Our key results are:

- We have verified and optimized more than 15 synchronization primitives from both the literature and industry – most of which are formally verified on WMMs for the first time.
- VSync discovered the following previously unknown bugs:
  - An await violation bug in the MCS lock of DPDK [18], reported and fixed in [5]. This bug exemplifies the difficulty in reasoning about WMM. Despite being a single-line bug fix, the discussion with the ARM engineers extended over 3 months until the patch was accepted.
  - A mutual exclusion violation bug in the CLH lock of seL4 [22], reported and fixed in [6]. The seL4 is a flagship of formal verification. The bug was in one of the few components that their verification did not cover, but which is extremely critical: the big kernel lock. This shows that VSync can complement functional formal verification as applied in seL4.
- VSYNC provides barrier optimizations comparable to experts, in a fraction of time: while experts optimized the barriers of Linux qspinlock [15] over several iterations over the course of years, VSYNC finds a comparable barrier combination within 11 minutes.

References

[1] Qspinlock code at version 4.4 of Linux Kernel. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/kernel/locking/qspinlock.c?h=v4.4.

[2] Qspinlock code at version 5.6 of Linux Kernel. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/kernel/locking/qspinlock.c?h=v5.6.

[3] spin_unlock optimization(i386), 1999. https://marc.info/?l=linux-kernel&m=94318921016232&w=2.

[4] rmem: Executable concurrency models for ARMv8, RISC-V, Power, and x86, 2009. https://github.com/rems-project/rmem.

[5] Await termination violation bug fix in DPDK, 2020. http://patches.dpdk.org/patch/75983/.

[6] Mutual exclusion bug fix in seL4, 2020. https://github.com/seL4/seL4/pull/199/commits.

[7] Parosh Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Phong Ngo. Context-Bounded Analysis for POWER. pages 56–74, 03 2017.

[8] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. Stateless model checking for TSO and PSO. *Acta Informatica*, 54(8):789–818, 2017.

[9] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. Stateless model checking for POWER. In *International Conference on Computer Aided Verification*, pages 134–156. Springer, 2016.

[10] Sarita V. Adve and Mark D. Hill. Weak ordering—a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, page 2–14, New York, NY, USA, 1990. Association for Computing Machinery.

[11] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence. In *International Conference on Computer Aided Verification*, pages 508–524. Springer, 2014.

[12] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(2):1–74, 2014.

[13] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Robustness against relaxed memory models. In *Software Engineering*, volume P-227 of *LNI*, pages 85–86. GI, 2014.

[14] Sebastian Burckhardt. Memory model sensitive analysis of concurrent data types. *Dissertations available from ProQuest*, 01 2007.

[15] Jonathan Corbet. locks and qspinlocks. https://lwn.net/Articles/590243/, 2014.

[16] Will Deacon. locking/qspinlock: Ensure node is initialized before updating prev->next, Feb 13, 2018. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=95bcade33a8a.

[17] Shaked Flur, Kathryn E Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 608–621, 2016.

[18] Linux Foundation. Data Plane Development Kit (DPDK), 2015.

[19] Gerard J Holzmann and William Slattery Lieberman. *Design and validation of computer protocols*, volume 512. Prentice hall Englewood Cliffs, 1991.

[20] Huawei. Huawei Unveils Industry's Highest-Performance ARM-based CPU, Jan 2019. https://www.huawei.com/en/news/2019/1/huawei-unveils-highest-performance-arm-based-cpu.

[21] Bengt Jonsson. State-space exploration for concurrent algorithms under weak memory orderings: (preliminary version). *SIGARCH Comput. Archit. News*, 36(5):65–71, June 2009.

[22] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, 2009.

[23] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Effective lock handling in stateless model checking. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–26, 2019.

[24] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Model checking for weakly consistent libraries. In *Proceedings of the 40th ACM SIGPLAN*

Conference on Programming Language Design and Implementation, PLDI 2019, pages 96–110, New York, NY, USA, 2019. Association for Computing Machinery.

[25] Michalis Kokologiannakis and Konstantinos Sagonas. Stateless model checking of the Linux kernel's hierarchical read-copy-update (tree RCU). In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, pages 172–181, 2017.

[26] Michalis Kokologiannakis and Viktor Vafeiadis. Hmc: Model checking for hardware memory models. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1157–1171, 2020.

[27] Michael Kuperstein, Martin Vechev, and Eran Yahav. Automatic inference of memory fences. *SIGACT News*, 43(2):108–123, June 2012.

[28] Nian Liu, Binyu Zang, and Haibo Chen. No barrier in the road: a comprehensive study and optimization of arm barriers. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 348–361, 2020.

[29] Waiman Long. locking/qspinlock: Use _acquire/_release() versions of cmpxchg() & xchg(), Nov 10, 2015. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=64d816cba06c.

[30] RISC-V. The risc-v instruction set manual. https://content.riscv.org/wp-content/uploads/2019/06/riscv-spec.pdf. Accessed: 2020-03-06.

[31] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. X86-tso: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, July 2010.

[32] Techcrunch.com. Microsoft updates its Arm-based Surface Pro X tablet with a faster CPU, 2020. https://techcrunch.com/2020/10/01/microsoft-updates-its-arm-based-surface-pro-x-tablet-with-a-faster-cpu/.

[33] The Guardian. Apple ditches Intel for ARM processors in Mac computers with Big Sur, 2020. https://www.theguardian.com/technology/2020/jun/22/apple-ditches-intel-for-arm-processors-in-big-sur-computers.

[34] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 209–220, 2015.

[35] Pan Xinhui. locking/qspinlock: Use atomic_sub_return_release() in queued_spin_unlock(), Jun 3, 2016. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=ca50e426f96c.

[36] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA+ specifications. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 54–66. Springer, 1999.