VSync: Push-Button Verification and Optimization for Synchronization Primitives on Weak Memory Models

Jonas Oberhauser
Huawei Dresden Research Center
Germany
Huawei OS Kernel Lab
China

Rafael Lourenco de Lima Chehab
Huawei Dresden Research Center
Germany
Huawei OS Kernel Lab
China

Diogo Behrens
Huawei Dresden Research Center
Germany
Huawei OS Kernel Lab
China

Ming Fu*
Huawei Dresden Research Center
Germany
Huawei OS Kernel Lab
China

Antonio Paolillo
Huawei Dresden Research Center
Germany
Huawei OS Kernel Lab
China

Lilith Oberhauser
Huawei Dresden Research Center
Germany
Huawei OS Kernel Lab
China

Koustubha Bhat
Huawei Dresden Research Center
Germany
Huawei OS Kernel Lab
China

Yuzhong Wen
Huawei OS Kernel Lab
China

Haibo Chen
Huawei OS Kernel Lab
Shanghai Jiao Tong University
China

Jaeho Kim
Huawei Dresden Research Center
Germany
Huawei OS Kernel Lab
China

Viktor Vafeiadis
Max Planck Institute for Software
Systems (MPI-SWS)
Germany

ABSTRACT

Implementing highly efficient and correct synchronization primitives on modern *Weak Memory Model* (WMM) architectures, such as ARM and RISC-V, is very difficult even for human experts. We introduce VSync, a framework to assist in optimizing and verifying synchronization primitives on WMM architectures. VSync automatically detects missing and overly-constrained barriers, while ensuring essential safety and liveness properties. VSync relies on two novel techniques: 1) *Adaptive Linear Relaxation* (ALR), which utilizes barrier monotonicity and speculation to quickly find a correct maximally-relaxed barrier combination; and 2) *Await Model Checking* (AMC), which for the first time makes it possible to check termination of await loops on WMMs.

We use VSync to automatically optimize and verify state-of-the-art synchronization primitives from systems like seL4, CertiKOS, musl libc, DPDK, Concurrency Kit, and Linux, as well as from the literature. In doing so, we found three correctness bugs on deployed systems due to missing barriers and several performance bugs due to overly-constrained barriers. Synchronization primitives optimized by VSync have similar performance to industrial libraries optimized by experts.

CCS CONCEPTS

• **Theory of computation** → **Verification by model checking**; • **Software and its engineering** → *Software testing and debugging*; • **Computer systems organization** → *Multicore architectures.*

KEYWORDS

model checking; weak memory models

ACM Reference Format:
Jonas Oberhauser, Rafael Lourenco de Lima Chehab, Diogo Behrens, Ming Fu, Antonio Paolillo, Lilith Oberhauser, Koustubha Bhat, Yuzhong Wen, Haibo Chen, Jaeho Kim, and Viktor Vafeiadis. 2021. VSync: Push-Button Verification and Optimization for Synchronization Primitives on Weak Memory Models. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3445814.3446748

1 INTRODUCTION

Modern multicore architectures, such as ARM, Power, and RISC-V, follow *weak memory models* (WMMs) [18, 35, 81, 89], which allow them to execute independent memory operations out of order. As these WMMs are becoming increasingly pervasive (*e.g.*, Huawei Kunpeng servers [44] and recent Apple computers [85] run on

[*] Ming Fu (ming.fu@huawei.com) is the corresponding author.

ASPLOS '21, April 19–23, 2021, Virtual, USA

ACM ISBN 978-1-4503-8317-2/21/04.
https://doi.org/10.1145/3445814.3446748

ARM), a lot of concurrent software designed for older, fairly strong memory models such as SPARC/x86 TSO [82, 83] need to be ported to these modern WMMs.

The good news is that most software uses only synchronization primitives for inter-thread communication; provided synchronization primitives are correct, such software work on WMMs out of the box [15]. 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 are often the only means of inter-thread communication, they lie on the critical path, and unnecessary barriers in synchronization primitives affect the performance of the complete system [1]. For this reason, experts spend a lot of time and effort in identifying the key memory operations that need to be executed in order, and optimizing the usage of barriers accordingly (*e.g.*, [26, 63–65, 90]).

Naturally, ensuring efficiency and correctness of these primitives gets restricted to only a small group of experts. Although this is deemed acceptable, our evaluation of several synchronization primitives implemented and optimized by experts for ARM and other WMMs shows that even experts are prone to getting it wrong, introducing either unnecessary barriers, or worse: subtle data races and hangs. Due to non-determinism introduced by concurrency, and even more so by WMMs, these problems are hard to reproduce and debug. As synchronization primitives are generally assumed to be correct by programmers, it is also hard to trace these problems back to the synchronization primitives. For these reasons, it is our opinion that selecting barriers to ensure performance and correctness of synchronization primitives on WMMs is not a task that should be left to humans.

In this paper, we present VSYNC, a framework that automates this task of ensuring performance and correctness of synchronization primitives on WMMs. It takes a synchronization primitive implemented with possibly overly-constrained barriers (*e.g.*, sequentially consistent barriers on every memory operation) and optimizes them to a maximally-relaxed yet correct barrier combination. We achieve this by strategically relaxing the barriers (precisely, relaxing the barrier modes) and using a *model checker* [46] to check each relaxation.

We encountered two main challenges. First, the state space of possible barrier combinations is huge (exponential in the size of the program) and so naively searching through it (*e.g.*, with a breadth-first-search) is hopeless. We overcome this challenge by exploiting the monotonicity [87] of barrier relaxations—namely, *relaxing an already incorrect barrier combination can never produce a correct one*. We develop the *Adaptive Linear Relaxation* (ALR) algorithm, which gradually relaxes one barrier at a time until no further correct relaxation is possible, and uses adaptive timeouts to guide the search towards a solution. This process invokes the model checker only a polynomial number of times, and still produces a maximally-relaxed barrier combination.

Second, state-of-the-art model checkers cannot automatically detect one key class of bugs introduced by WMMs: hangs. As a result, simply running the ALR algorithm with existing model checkers for WMMs often yields incorrect barrier combinations. The reason for this limitation is quite fundamental. The state-of-the-art in model checking for WMMs is *stateless model checking* (SMC) [10, 13, 53–56], which examines individual execution traces of bounded length without recording the set of visited states and so cannot, in general, reason about the (non)-termination of programs[1]. Nevertheless, in all the synchronization primitives we have examined, hangs are caused exclusively by *await loops*: loops that do not have any side-effects except perhaps for their final iterations. Noting that reasoning about the termination of await loops is substantially easier than the general case, we develop *Await Model Checking* (AMC), which extends SMC with the ability to prove termination of await loops (await termination, AT), and show that it is sufficient for verifying liveness of synchronization primitives.

```
              locked = 0, q = 0;

       T1 : lock               ||   T2 : unlock
locked = 1;                    ||  while (q == 0);
q = 1;                         ||  locked = 0;
while (locked == 1);           ||
/* Critical Section */         ||
```

Figure 1: Await loops in one path of an MCS lock. T_1 signals `q = 1` to notify T_2 that it enqueued. Once T_2 receives the notification it signals back `locked = 0` to pass the lock to T_1.

We illustrate both challenges with a greatly simplified portion of the MCS lock [40, 70] shown in Fig. 1. Two threads T_1 and T_2 use await loops to synchronize: T_2 waits for `q` to change from `0`, as a signal that T_1 has been enqueued. T_1 waits for `locked` to change from `1`, as a signal that T_2 has released the lock. Without any barriers placed, on ARM, Power and RISC-V, the write operations of T_1 can be executed out of order. In this case T_2 may receive the signal `q = 1`, leave its loop, and write `locked = 0` before T_1 writes `locked = 1`. Then T_1 never observes the value `locked = 0` that it overwrites, and is stuck in the loop forever. Even for this tiny example, there are 5 possible places for inserting barriers (one for each memory access) and if the architecture has 3 different modes of barriers, we have a search space of $(3 + 1)^5 = 1024$ possible combinations.

As case studies, we used VSYNC to formally verify and optimize synchronization primitives on WMMs, some of which were implemented by experts for WMMs, and some of which we implemented ourselves from literature. Those include: five locks from Herlihy and Shavit's textbook [40], the big kernel CLH lock of seL4 [51, 79], the MCS lock in CertiKOS [38, 39, 50], DPDK [36], Concurrency Kit [16], and in an internal Huawei product (ported from x86 to ARM), the TWA lock [28], the mutex in musl libc [7], Linux's qspinlock [25] and NUMA-aware locks [30].

Most of these synchronization primitives are formally verified on WMMs for the first time. VSYNC reported 3 barrier bugs: an overly-relaxed barrier in the seL4 lock, a missing barrier in the ported MCS lock and an overly-relaxed barrier in the DPDK MCS

[1] There exist also some *stateful* model checkers for WMMs [12, 21, 42, 47, 57, 81, 91], which in principle can detect hangs (though most implementations do not actually do so). Sadly, however, they do not scale: they run out of memory on non-trivial programs such as synchronization primitives.

Table 1: Mapping from IMM/VSync to RISC-V and ARMv8. For ARMv8.1 and later, read-modify-write operations can also map to specialized instructions, *e.g.*, `atomic_xchg` to `SWPAL`.

VSync	RISC-V	ARMv8
`atomic_xchg`	`amoswap.w.aqrl`	`LDAXR;STLXR`
`atomic_xchg_acq`	`amoswap.w.aq`	`LDAXR;STXR`
`atomic_xchg_rel`	`amoswap.w.rl`	`LDXR;STLXR`
`atomic_xchg_rlx`	`amoswap.w`	`LDXR;STXR`
`atomic_read`	`fence [rw,rw]; lw; fence [r,rw]`	`LDAR`
`atomic_read_acq`	`lw; fence [r,rw]`	`LDAR`
`atomic_read_rlx`	`lw`	`LDR`
`atomic_write`	`fence [rw,w]; sw; fence [rw,rw]`	`STLR`
`atomic_write_rel`	`fence [rw,w]; sw`	`STLR`
`atomic_write_rlx`	`sw`	`STR`
`atomic_fence`	`fence [rw,rw]`	`DMB SY`
`atomic_fence_acq`	`fence [r,rw]`	`DMB LD`
`atomic_fence_rel`	`fence [rw,w]`	`DMB SY`
`atomic_fence_rlx`	`NOP`	`NOP`

lock. The first two bugs compromise mutual exclusion of critical sections. The bug in the DPDK MCS lock can potentially cause hangs. The three bugs have been confirmed by the maintainers and our patches have been merged.

Moreover, the synchronization primitives optimized by VSync have similar performance as industrial libraries optimized by experts. Even for complex code such as Linux's qspinlock we quickly obtain similar results to the implementation optimized by experts.

Contributions. The contributions of this paper are:

- The VSync framework, which automatically optimizes barriers in synchronization primitives on WMMs to a provably correct and maximally-relaxed combination (§2.2).
- Adaptive Linear Relaxation (ALR), an efficient barrier optimization algorithm based on speculation (§3).
- Await Model Checking (AMC), which for the first time allows stateless model checkers to verify termination of side-effect-free loops on WMMs (§4).
- A set of provably-correct and high-performance synchronization primitives for practical use in industry (§5).
- A report of our experience in using VSync in several open-source projects and an internal Huawei product (§6).

2 BACKGROUND AND OVERVIEW OF VSYNC

2.1 Weak Memory Models

Barriers and modes. To improve performance, modern architectures apply optimizations such as store buffering and load buffering [86]. These optimizations can cause observable memory access reorderings in concurrent programs and introduce subtle bugs [23, 87]. To write correct code, programmers need to study the often complex WMM specification of their target platform, which defines the visible effects allowed by the optimizations, and also how to (locally) turn them off as needed by means of so called *barriers*.

The VSync atomics library offers two types of barriers (see Table 1): *implicit barriers*, *i.e.*, barriers attached to atomic memory operations; and *fences*, also called *explicit barriers*, *i.e.*, stand-alone barriers. We consider four common modes of barriers: *rlx* (*relaxed*), *acq* (*acquire*), *rel* (*release*), and *sc* (*sequentially consistent*). They allow different levels of optimizations, and their relation is illustrated in Fig. 2. Modes appear in the suffix of each barrier, *sc* being the default mode. The precise meaning of the modes depends on the WMM, but intuitively, the relaxed mode *rlx* allows all optimizations whereas the sequentially consistent mode *sc* allows no (visible) optimizations. The modes *rel* and *acq* are in-between and are used to ensure ordering across threads while still providing good performance; for example, in Fig. 1 on ARMv8, a barrier with *rel* mode on `q = 1` can be used to ensure `locked = 1` of T_1 will be performed before `locked = 0` of T_2, and not the other way around.

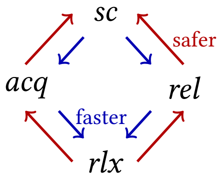


Figure 2: Partial order of barrier modes.

VSync allows users to plug in modules implementing the WMMs of the target platforms. It supports not only traditional memory models, such as SPARC/x86 TSO [82] and PSO [83], but also the more advanced dependency-tracking memory models [56], such as ARMv7 [18], ARMv8 [35], RISC-V [89], PowerPC [68], and the Linux-kernel memory model (LKMM) [3]. Currently, VSync only includes a module for the intermediate memory model (IMM) [80], which unifies most dependency-tracking WMMs with provably-correct barrier mappings. Table 1 lists a few mappings from VSync atomics to RISC-V and ARMv8.

Impact of barriers. Synchronization primitives employ many barriers and often multiple awaits (see Table 2 on Page 8). Awaits are either used directly by the primitive, *e.g.*, for waiting on other threads to leave the critical section; or they are used by VSync to model kernel calls such as `futex_wait()` which prevents the thread from running until a condition is met. Each await operates quite differently, and AT (await termination) cannot be guaranteed by placing barriers on the polled variable; *e.g.*, to ensure AT of T_1, which polls `locked` in Fig. 1, barriers need to be added to `q`; without these barriers, T_1 will be forever spinning on `while(locked == 1)`, even if all accesses to `locked` use *sc*-mode. For this reason, finding the correct barriers to ensure AT is challenging and cannot be solved by pattern-matching, or other simple means.

On the one hand, not finding these barriers results in hangs on real hardware, which we have experienced, *e.g.*, with the TWA lock and the MCS lock. Moreover, with insufficient barriers, mutual exclusion of the synchronization primitives is violated; this compromises the integrity of the critical section, resulting in data races and corrupted data, *e.g.*, in the subtle bug we found in the CLH lock of seL4 (discussed in details in §6). On the other hand, over-constraining tremendously hurts performance. Table 3 on Page 9 shows an excerpt of our kernel-level benchmark with these synchronization primitives. By replacing *sc* barriers with more relaxed barriers, we achieve considerable speedup. This is not unexpected:

```
1 typedef atomic_t lock_t;
2 void lock_acquire(lock_t *lock) {
3   do {
4     while(atomic_read(lock)) { }
5   } while(atomic_xchg(lock, 1));
6 }
7 void lock_release(lock_t *lock) {
8   atomic_write(lock, 0);
9 }
```

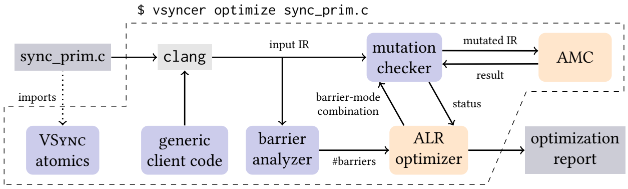
Figure 3: Example TTAS lock implementation

```
optimization report
-------------------

lock_acquire
  atomic_read --> rlx
  atomic_xchg --> acq

lock_release
  atomic_write --> rel
```

Figure 4: VSync optimization report



```
$ vsyncer optimize sync_prim.c
```

Figure 5: `vsyncer optimize` reports the maximally-relaxed barrier-mode combination of a synchronization primitive.

removing just one unnecessary barrier in the Linux-spinlock improved overall performance of the Linux-kernel by 4% [1].

2.2 The VSync Framework

The VSync framework allows users to verify and optimize synchronization primitives. To explain the main components of VSync, we consider the example of a user implementing the test-and-test-and-set (TTAS) lock, following the literature [40]. The implementation is depicted in Fig. 3. To acquire the lock, a thread atomically exchanges the lock value with 1 (Line 5). If the old value is 0, the thread enters the critical section, otherwise keeps retrying. To avoid excessive cache invalidation, the thread first waits for the lock to be released before trying to acquire it (Line 4). To release the lock, the thread sets its value to 0 (Line 8). Being a well-known algorithm, the user assumes the code is correct on sequential consistency. Now the user wants to optimize the barriers.

The user calls the `vsyncer optimize` command (see Fig. 5), providing the TTAS lock implemented with VSync atomics in Table 1. Note that atomic operations without suffix have *sc* barrier modes. A generic client code is automatically linked together for the subsequent process. VSync outputs a barrier-optimization report (Fig. 4), which suggests relaxations, *e.g.*, modifying Line 5 to use an *acq* barrier mode as in `atomic_xchg_acq()`. To produce the optimization report, the ALR optimizer in VSync gradually relaxes the barriers, while deploying the AMC to verify the correctness of the current barrier combination. The suggested barriers in the report are guaranteed to be correct and *maximally-relaxed*, *i.e.*, relaxing any of the barriers further causes the implementation to fail on the WMM defined by the currently selected module.

Generic client code. Model checking requires a *client code*, *i.e.*, a main function which generates some number of threads that call `lock_acquire()` and `lock_release()` functions, access shared variables, and so on. Model checking can only verify the correctness of a library with respect to that client code. We provide a client code for synchronization primitives which is parametric in the number N of threads. These threads access a critical section through the synchronization primitive and increment a shared variable. The mutual exclusion is verified by asserting an expected value of the shared variable. Liveness is guaranteed by AMC, as discussed later. It is up to the user of VSync to find a sufficiently large parameter N for which the client code can fully exercise the code and thus produce a dependable verification and optimization result.

We compile the synchronization primitives along with the generic client code with `clang` [6] into an LLVM intermediate representation (IR) module, called *input IR* in Fig. 5.

Barrier analyzer and ALR optimizer. We analyze the input IR module for the position and number of barriers. This information is passed to the ALR optimizer, which then starts the optimization as described in §3. Once finished, the optimizer returns a report to the user similar to Fig. 4.

Mutation checker and AMC. The optimizer decides the next barrier-mode combination to be checked; AMC performs the actual check (§4). The mutation checker intermediates the process by *mutating* the input IR according to the desired barrier-mode combination and subsequently calling AMC. AMC returns *success* if the *mutated IR* is correct and *failed* otherwise. The mutation checker forwards the check status to the optimizer. In the following sections, we focus on the ALR optimizer and AMC components.

3 ADAPTIVE LINEAR RELAXATION

We present an efficient algorithm to maximally relax correct input barrier-mode *combinations* for a given synchronization primitive. A barrier-mode *combination* b is an array representing the barrier modes of the synchronization primitive. If the synchronization primitive, with the modes in combination b, satisfies mutual exclusion and termination, we say b is *correct*. If any further relaxation of b is incorrect, we say b is *maximally-relaxed*. We highlight this important definition:

Definition 1 (Maximally-relaxed combination). *A combination b is maximally-relaxed iff 1) b is correct, and 2) there is no correct combination b' such that $b' \neq b$ and $b'[i] \sqsubset b[i] \lor b'[i] = b[i]$ for all $i \in \{1, \ldots, |b|\}$, where $rlx \sqsubset acq \sqsubset sc$, $rlx \sqsubset rel \sqsubset sc$, and $|b|$ is the length of b.*

Note that there can be multiple maximally-relaxed combinations; we only find one of them. In general, finding all maximally relaxed solutions is intractable, as their number is in the worst case exponential in the number of barriers. This is in particular the case when using redundant fences (*i.e.*, explicit barriers), for which usually there are multiple correct positions. For example, if two fences are inserted in each of n lines, VSync can in each line pick either the first or the second fence, leading to $2^{n/2}$ combinations. In practice, most synchronization primitives have exactly one maximally-relaxed barrier combination if no fences are being used; this is true for all except two of the synchronization primitives studied in this paper. Even for the remaining two, the performance difference between the different maximally relaxed combinations is negligible.

Linear Relaxation (LR). Algorithm 1 depicts our base algorithm, LR. One-by-one, LR relaxes each barrier $b[i]$, picking the

```
function LR()
    $b \leftarrow [sc, \dots, sc]$
    for $i \in [1, \dots, |b|]$ do
        for $m \in [rlx, acq, rel, sc]$ do
            $b[i] \leftarrow m$
            if $b[i] = sc \lor$ check($b$) then
                break
    return $b$
```

Algorithm 1: Linear Relaxation (LR)

```
function ALR()
    $\tau \leftarrow 10ms$
    while true do
        $b \leftarrow \mathrm{LR}^{\tau}()$
        if $b = [sc, \dots, sc] \lor$ check($b$) then
            return $b$
        $\tau \leftarrow \tau$ + time taken to check $b$
```

Algorithm 2: Adaptive LR (ALR)

	$b[1]$	$b[2]$	$b[3]$	check(b)
iterations ↓	***rlx***	*sc*	*sc*	true
	rlx	***rlx***	*sc*	false
	rlx	***acq***	*sc*	true
	rlx	*acq*	***rlx***	false
	rlx	*acq*	***acq***	false
	rlx	*acq*	***rel***	true

Figure 6: Linear Relaxation of TTAS lock example. LR() = [*rlx*, *acq*, *rel*]

most relaxed mode (with the ordering from Fig. 2) that is still correct. LR finds this mode by trying each possibility from weakest (*rlx*) to strictest (*sc*), stopping as soon as the combination b is correct – in the worst case when $b = [sc, \dots, sc]$. Figure 6 shows the execution of LR for the TTAS lock (Fig. 3), which has 3 barriers. The output of LR() is [*rlx*, *acq*, *rel*], which is maximally relaxed. At the core of LR is the check(b) function, which calls AMC underneath after mutating the barriers of the synchronization primitive according to the combination b. If the mutated synchronization primitive is incorrect, or if some barrier mode in b is not applicable to the actual barrier in the code, check(b) returns false; otherwise check(b) returns true, showing that the mutation with b is correct.

Speculative checks. For each barrier in b, LR makes up to three calls to check(b), at most one of which returns true. To return true, AMC has to inspect every execution, while to return false, AMC can stop as soon as it finds the first execution that triggers a bug. Therefore, checks that return true are usually slower than checks that return false (up to two orders of magnitude in our experience, see §5). To avoid these up to $|b|$ highly expensive calls, we introduce an adaptive timeout mechanism, which speeds up the gradual relaxation. We speculatively accept barrier-mode combinations as correct if AMC takes longer than a timeout τ to check them. For this, we use the function check$^{\tau}(b)$, which returns true if b is correct or if AMC takes longer than τ to terminate. We define LR$^{\tau}$ to be Algorithm 1 with the speculative check function check$^{\tau}$ called in place of function check.

Adaptive timeouts. Selecting the right timeout is crucial for successful and efficient exploration, but cannot be done *a priori*. Too large values of τ hinder the convergence of the search, while too small values can lead LR$^{\tau}$ to return overly-relaxed combinations.

To solve this, our *Adaptive Linear Relaxation* (ALR) algorithm in Algorithm 2 starts with a low value of τ (*e.g.*, 10ms), and increases it until a correct combination is produced. In more detail, ALR calls LR$^{\tau}$ to find a speculatively-correct combination b, which is then fully checked without a timeout. If combination b is really correct, it must be maximally relaxed and is returned. If combination b is actually incorrect, ALR restarts the search afresh with a τ value slightly larger than the time AMC took to check b.

With ALR, the time to produce a correct and maximally-relaxed barrier combination for the Linux qspinlock takes three minutes (instead of several hours with LR and no timeouts); subsequently verifying that this combination is indeed correct without timeouts takes another eight minutes.

4 AWAIT MODEL CHECKING

As discussed in §1, the state-of-the-art in verification on WMMs is *stateless model checking* (SMC) [10, 13, 14, 27, 43, 52, 54–56, 71]; that is, enumerating all possible executions of a concurrent program subject to certain optimizations such as dynamic partial order reduction (DPOR) [34]. By design, however, SMC only works for programs that have a finite state space (*i.e.*, only a finite number of executions) and terminate unconditionally (*i.e.*, each execution is of finite length). As a result, SMC cannot soundly handle programs with *await loops* because their termination on WMMs often depends on the loops not continuously reading stale values (*e.g.*, Fig. 1). As such, there is no *a priori* bound on the number of iterations such loops can observe stale values.

Await loops are ones whose execution is free of side-effects except perhaps in their last iteration. They are extremely common in synchronization primitives, *e.g.*, when waiting for a resource to become available. Without sufficient barriers, under WMM, such loops can repeatedly read stale values, which can cause hangs. In the synchronization primitives we have studied, they are the only potential source of hangs. Await loops can be manually annotated or easily identified by static analysis [33], and it is even common for model checkers to convert simple await loops into `assume` statements, but, as we will see in §4.4, such a conversion is sound only for checking safety properties, not for checking their termination.

In §4.1, we present our assumptions about programs; in §4.2, we present our approach for checking termination of await loops; and in §4.4 we show why seemingly similar approaches from the literature do not solve the problem.

4.1 Applicable Domain

We assume programs that satisfy the following two conditions: 1) awaits are side-effect-free except in their final iteration, and 2) only awaits can cause them to hang. We make this precise by the following two principles.

Bounded-Effect Principle: Side-effects of non-final iterations of an await must be contained within that iteration: they can only write to local variables (*e.g.*, registers), whose values must not be read by subsequent await iterations or by any code thereafter. In contrast, the final iteration of an await is allowed to have visible side-effects, *e.g.*, by performing a compare-and-swap operation which acquires a resource.

Bounded-Length Principle: There is a constant integer C such that if all awaits are unrolled once, all executions of the program must have at most C steps.

These conditions are satisfied by all synchronization primitives we have studied as well as by all programs for which existing stateless model checkers are sound. Thus, they do not restrict the domain of WMM model checkers, but rather extend it to programs with non-terminating awaits. Existing model checkers require that programs are loop-free (*e.g.*, [54, p. 97]) or have acyclic, finite state spaces (*e.g.*, [74, p. 22]), which are stronger conditions than our Bounded-Length principle.

4.2 Checking Termination of Await Loops

Our technique for proving (non-)termination of await loops is made possible by two key insights. The first is that infinite executions caused by non-terminating awaits can be represented as finite executions, which we call *stagnant*.

Definition 2. *A (possibly partial) execution is* stagnant *iff all remaining running threads are inside an await loop that they cannot exit based on the values currently available to them.*

Note that we consider here partial executions, because complete stagnant executions are infinite and thus cannot be fully generated by a model checker. Instead of waiting until the execution becomes infinite, this definition captures the moments in which it is foreseeable that the execution can be extended to an infinite one.

Perhaps counterintuitively, in some WMMs such as C11, partial stagnant executions can sometimes be extended to terminating executions due to so-called out-of-thin-air behaviors [59, p. 7]: *e.g.*, one of the running threads might (in such WMMs) speculate that it will eventually leave the await and speculatively execute a write that lies behind its await, which allows other threads to leave their loops and then produce writes that allow the original thread to really leave the loop and hence justify it's speculation.

However, even in those WMMs, each stagnant execution can also be extended to a non-terminating execution, by simply avoiding this kind of speculation. This forces threads to only generate writes that are inside their await loops. By the Bounded-Effect principle, these writes never produce new values for other threads, and hence the set of available values never changes. Since none of the available values allow leaving the loop, the threads are permanently stuck. This results in a non-terminating infinite execution.

Conversely, if we have an infinite execution, there is also a stagnant, finite execution. Recall that an execution is stagnant if all remaining running threads are inside an await and none of the available writes let them break the loop. Indeed, we will trim the infinite execution to a finite execution in which each remaining running thread has only *one* available write for each location, and reading those writes does not allow the threads to exit the loop. Observe first that the Bounded-Effect and Bounded-Length principles ensure that each thread only has a finite set of writes to read from. Standard WMMs guarantee that threads do not read a stale value forever (*e.g.*, progress axiom in [89, p. 88] and Sec. 7.17.3-16 in [45]). Since the set of writes is finite, this guarantee implies that a thread that is stuck inside an await must eventually read from the most up-to-date writes on each location. We trim the infinite execution of that thread to the point where those most up-to-date writes become the only available writes; by doing this for each thread, we obtain as promised a stagnant, finite execution.

Thus a stagnant finite execution exists iff an infinite execution exists. Exploiting this, a model checker that enumerates all finite executions of a program can detect non-terminating awaits by looking for stagnant executions. In Fig. 1 that would be an execution in which T_1 does not exit its `while` loop because `locked = 1` is the only value available to T_1.

The problem, however, is that enumerating all finite executions of a program with loops may not be possible, because the program itself can still have an infinite number of finite executions. Consider, for example, the expected execution order of Fig. 1 where T_2 writes last. After T_2 wrote `locked = 0`, T_1 is obviously allowed to observe this new value. However, on WMM it is also allowed to read the stale `locked = 1` any number $N = 1, 2, 3, \ldots$ of times [48]. This can, for instance, happen because the stale value remains in its cache for a non-deterministic (but finite) amount of time. Each value of N gives a different execution. Moreover, none of these executions is stagnant because while reading `locked = 1` in each iteration of the loop, the value `locked = 0`, which would allow the thread to exit the loop, remains available. A model checker might be stuck exploring this infinite number of non-stagnant executions and never reach the stagnant execution indicating the termination problem.

We address this problem with a second insight: if an execution that repeats the same await loop iteration twice in a row — reading the same values — results in an error, *e.g.*, a violated assertion, then an execution without such repetitions can also trigger the same error. Thus executions with such repetitions can be ignored. We call such executions *wasteful*.

Definition 3. *An execution is* wasteful *iff two consecutive iterations of an await read exactly the same values.*

AMC does not explore wasteful executions of the given program. However, it explores all other executions and reports an AT violation if any of them is stagnant. The correctness of AMC follows from the following theorem.

Theorem 1 (AMC Correctness). *Let P be a program satisfying the Bounded-Effect and Bounded-Length principles (§4.1). Then, the following properties hold:*

1. *P has a finite number of non-wasteful executions.*
2. *If P has an execution with a safety bug, then it also has an execution with the same bug that is not wasteful.*
3. *If P has an infinite execution, then it also has a stagnant execution that is not wasteful.*

Property 1 ensures that AMC terminates when enumerating all executions that are not wasteful. Intuitively, it holds because the Bounded-Length and Bounded-Effect properties imply that there is a bounded number of writes in each execution, and since consecutive iterations of each await loop of a non-wasteful execution have to read from a different write, the number of possible iterations is bounded. Property 2 says that enumerating non-wasteful executions suffices for finding safety bugs, while property 3 says that it suffices for reporting liveness bugs. Intuitively, these hold because of the Bounded-Effect property, which allows us to remove all but the last iteration of each await loop. We provide proofs in our technical report [77].

4.3 Summary of Implementation

Our AMC implementation generates executions incrementally, *i.e.*, by adding one read or write of one thread at a time. The main change over existing model checkers is that we never add reads that would result in a wasteful execution. We ensure this by adding a read to an await loop iteration only if either a) it is the first iteration of the loop, b) this read or a previous read of this iteration reads a value not read in the preceding iteration of this loop (hence ensuring that the execution is not wasteful no matter what subsequent read operations will read), or c) there is a later read in the iteration that can read a new value (*i.e.*, we can still avoid generating a wasteful execution later). As long as none of these options are possible (*e.g.*, because no values distinct from those read in the previous iteration are available), the exploration of that thread is paused. This can lead to a deadlock-like situation in which the model checker cannot explore anything because the exploration of all currently running threads is paused. It is easy to see that this situation describes a stagnant execution, and is thus proof of a non-terminating await. Conversely, it can be shown that every stagnant non-wasteful execution, if explored in this manner, will eventually lead to such a situation. This permits us to detect non-termination by checking whether such a situation occurs during exploration, without any explicit check for stagnancy.

4.4 Comparison to State-of-the-Art

In the literature there are some techniques that seem at first glance similar to our approach. However, the subtle differences prevent them from effectively detecting non-terminating loops. We now discuss the limitations of these techniques.

Stateful model checking. The first technique is to record all visited program states, which can be used to find cycles in the state space. This technique has been used so far only in model checkers for strong memory models [42, 91]. In theory, it can probably be extended to some (but not all) WMMs [11]. Even where applicable, however, this approach is impractical due to the huge number of reachable explicit program states.

Fairness. The second technique is *fairness* [71, 75], which ensures that executions cannot indefinitely read stale values. A model checker that uses fairness ensures that each await only reads a finite number of times from the same value, if a more up-to-date value is available or can be produced by another thread. This is an important step in the direction of detecting await termination, as an experienced verification engineer can inspect surprisingly long execution traces generated by these model checkers to look for hints that an await is not terminating. Unlike the set of executions that are non-wasteful, the set of fair executions that these model checkers explore is not finite. Thus, the model checker, on its own, cannot determine when it is safe to stop exploring. Furthermore, it cannot automatically detect whether a particularly long execution is an indicator for a non-terminating loop. In our technique, this is solved by checking if the execution is stagnant.

Bounded loop unrolling. The third technique is to unroll awaits a certain number of times [27, 37, 52, 54], which limits the exploration to a finite set of executions. Bounded loop unrolling produces (wasteful) executions in which each iteration of an await reads the same values. These executions seem at first glance stagnant; however, they may not be stagnant, because that would require that the reason for reading the same value over and over is that there is no other value that could be read, and that no such value will be produced by other threads in the future. In a finite execution, threads are allowed to read from the same variable arbitrarily often even in the presence of a more up-to-date value. Thus the wasteful executions generated by bounded loop unrolling are no indication of non-termination.

```
                 x = 0;
     T1    ||          T2
           ||
 x = 1;    || // while (x == 1);
           || assume (x != 1);
```

Figure 7: Using assume to model awaits

If the number of unrolled iterations is chosen sufficiently large, then every non-wasteful execution will be explored, and thus 1) no safety bugs are hidden by sufficiently large bounded loop unrolling, and 2) if one checks for each generated execution whether it is stagnant, one can use bounded loop unrolling to decide termination. However, many other redundant (*i.e.*, wasteful) executions will also be explored, resulting in sometimes drastic performance degradation (see Table 3 on Page 9, which compares the verification times of AMC, which only generates non-wasteful executions, and GenMC, which uses bounded loop unrolling with the minimal loop bound that is sufficient to verify safety). Finally, using bounded loop unrolling requires finding such a sufficiently large bound, either by the user or by some advanced automated technique; simply restricting exploration to non-wasteful executions does not require that extra effort.

Spin-assume transformation. The fourth technique is to reduce awaits to their final successful iteration [12, 54] using `assume` directives, similar to how purity [33] can be used for safety proofs. This restricts exploration to a strict subset of the executions that are non-wasteful. Unfortunately, this may eliminate all stagnant executions from the exploration. For example, consider the program in Fig. 7. After T_1 terminates, T_2 can on WMMs read either the stale value `x = 0` or the up-to-date value `x = 1` produced by T_1. With the assume directives, only the former execution will be generated as the latter does not satisfy the assumption, but the latter execution, which is a possible execution on WMMs and not wasteful, is stagnant. In other words, this technique hides bugs that cause await termination violations.

5 EVALUATION

We now evaluate different aspects of VSync: (1) Is the AT detection offered by AMC relevant in practice? (2) How does AMC compare to state-of-the-art model checkers for WMM? (3) How effective and efficient is the ALR barrier optimization? (4) Are optimized synchronization primitives faster than their unoptimized counterparts? For details of the setup and more detailed results, we refer to our technical report [77].

Table 2: Synchronization primitives evaluated with VSync. Complexity: lines of code (LoC) at the LLVM-IR level, number of await loops (#A), and number of barriers (#B) including atomics operations and fences. Verification time: time needed to verify a correct combination of the lock. Requires AT detection: primitives that may hang if barriers are too relaxed; AMC is capable of detecting it. Optimization time: time of running LR and ALR.

Synchronization Primitive	Complexity			Barrier modes after optimization				Requires AT detection?	Verification time (s)		Optimization time (s)	
	LoC	#A	#B	*sc*	*acq*	*rel*	*rlx*		AMC	GenMC	LR	ALR
CLH lock[40]	500	1	4	1	1	1	1		0.04	0.28	0.49	0.62
ArrayQ lock[40]	456	1	4	1	1	1	1		0.05	0.27	0.50	0.64
Ticketlock[2]	416	1	4	0	1	1	2		0.12	1.04	0.74	0.61
Semaphore	504	2	6	0	2	2	2		0.26	0.82	2.02	1.11
CertiKOS MCS[39]	954	2	9	2	1	1	5	✓	1.01	59.24	8.45	2.90
TWA lock[28]	581	2	10	0	4	2	4	✓	1.26	12.50	14.04	2.79
MCS lock[40, 70]	616	2	10	1	2	3	4	✓	1.57	69.95	15.40	2.95
CAS lock	431	1	3	0	1	1	1		6.73	99.01	21.14	7.43
RW lock	676	4	10	0	2	2	6		4.64	45.53	39.38	6.38
TTAS lock[40]	394	1	3	0	1	1	1		14.80	567.63	45.91	15.76
c-TKT-MCS[30]	979	4	27	1	4	6	16	✓	4.92	138.08	126.03	10.32
3-state mutex[31]	603	1	9	0	3	3	3	✓	28.30	127.95	163.12	30.62
rec. CAS lock	472	1	4	0	1	1	2		45.03	585.24	190.40	47.48
c-MCS-TWA	1140	4	31	1	6	7	17	✓	14.56	323.63	433.61	61.36
c-TTAS-MCS[30]	1017	4	26	1	4	6	15	✓	29.19	957.13	765.13	40.97
HCLH lock[40, 67]	907	2	12	2	4	1	5		344.26	9900.87	1891.42	464.93
qspinlock[25]	1160	5	26	1	7	2	16	✓	501.71	17116.62	8732.68	675.11
musl mutex[7]	683	1	14	0	3	2	9	✓	11936.16	74530.00	56901.40	12373.45

Synchronization primitives. We have applied VSync on several synchronization primitives from the literature and open source projects, listed in Table 2. The selection contains three groups of primitives: **spinlocks**, from simple TTAS lock to complex NUMA-aware cohort locks such as c-TKT-MCS [30] and Linux qspinlock [25]; 2 futex-based **mutexes**, one being the algorithm implemented in musl libc [7]; 2 **reader-writer locks**, one unfair (Semaphore) and one writer-preferring implementation (RW lock). We also introduce a new cohort lock, c-MCS-TWA, to exercise the ALR optimization with a complex algorithm combination. In Table 2, LoC refers to the numbers of lines of LLVM-IR code. We report the results with $N = 3$ threads (cf. §2.2) for all synchronization primitives – VSync finds the same optimizations with $N = 4$.

Implementation. We developed AMC in C++ as an extension to GenMC [54], supporting VSync and C11 atomics. Our experiments employ the IMM memory model of GenMC. We built our ALR optimizer and the tools around it in Golang.

5.1 On the Necessity of AT Detection

The column “*Requires AT detection*" in Table 2 marks synchronization primitives for which we found overly-relaxed barrier combinations that cause hangs, *i.e.*, executions with non-terminating await loops. Techniques such as loop unrolling and spin-assume transformation (§4.4) limit how often await loops iterate. Above this limit, the execution is considered *correct* even though it hangs on real hardware. In contrast, AMC detects such executions as *stagnant*, properly identifying the barrier combination as overly relaxed.

We emphasize that await termination (AT) detection is crucial for correct optimization. With a model checker that cannot detect AT, automatic barrier optimization invariably over-relaxes barriers, causing several primitives to hang. We experienced this firsthand when we used GenMC.

Note that we have not exhaustively checked all barrier combinations looking for overly-relaxed combinations that trigger hangs. Therefore, non-marked synchronization primitives are not guaranteed to terminate for any barrier combination.

5.2 Verification Performance

Although no other WMM model checkers can verify AT, we compare the verification time of AMC and other tools, such as GenMC. Our experiments run on a Workstation with a 12-core Intel Xeon W-2133 at 3.60GHz and 16GB RAM. Table 2 (*Verification time*) shows the time taken to verify a correct barrier combination for each synchronization primitive.

AMC verifies the primitives up to two orders of magnitude faster than GenMC, *e.g.*, 501s versus 17116s for qspinlock. The reason for the massive difference is that AMC explores no wasteful executions, whereas GenMC unrolls await loops as often as the selected limit, even if the values read in each iteration of the loop have not changed. For these experiments, we selected an unroll limit of 4, which is the smallest number that avoids hiding safety bugs. With smaller unroll values, loops bounded by the thread number $N = 3$ are not fully executed, *e.g.*, in initialization loops.

Note that the verification of musl mutex took considerably longer than for other synchronization primitives – more than 3 hours with AMC and more than 20 hours with GenMC. For example, in contrast to the 3-state mutex, musl mutex has an additional counter for waiters, which is incremented and decremented every time a thread fails to acquire the mutex. The multiple orders in which this

Table 3: Speedups of VSync-optimized synchronization primitives over their *sc*-only variants, evaluated with microbenchmarks (see Section 5.4.1). For each synchronization primitive, aggregated values are reported (max, min, median, mean, and standard deviation), and then more detailed results, per number of threads, are provided. For each "*t* thr." column, the reported values corresponds to $\frac{T_o}{T_s} - 1$, where T_o is median throughput of VSync-optimized and T_s is the median throughput of *sc*-only variants. Medians are computed over 5 runs with *t* threads, each run lasting for 30 seconds. Missing values ("-" in the table) corresponds to unstable results, filtered-out of the final results.

Synchronization	Aggregated speedups (%)					Speedups of median throughput per thread count (%)									
Primitive	max	min	median	mean	std	1 thr.	2 thr.	4 thr.	8 thr.	16 thr.	23 thr.	31 thr.	63 thr.	95 thr.	127 thr.
CLH lock[40]	33	-1	11	12	9	33	15	16	7	5	6	15	10	13	-1
ArrayQ lock[40]	26	14	19	20	4	19	21	18	26	23	17	16	14	19	23
Ticketlock[2]	16	-6	0	1	6	5	-2	0	-1	0	0	-2	-6	0	16
Semaphore	11	-9	0	0	6	11	-1	-	0	5	0	0	-6	-9	-
CertiKOS MCS[39]	74	1	7	15	22	74	12	8	16	21	3	5	2	6	1
TWA lock[28]	34	-4	-1	3	11	34	-3	-2	0	-2	-3	-4	1	3	3
MCS lock[40, 70]	78	-3	0	11	25	78	28	12	-1	0	-1	0	-3	0	-3
CAS lock	5	-6	-1	-1	3	5	-1	-	-6	0	-1	0	-2	-	-
RW lock	55	-41	-3	-1	29	30	-41	-	55	9	-3	-3	-27	-15	-11
TTAS lock[40]	9	-11	-2	-1	6	9	-2	-	-2	-1	-3	-3	2	-11	-
c-TKT-MCS[30]	63	5	28	32	20	63	17	5	13	29	41	47	18	27	57
rec. CAS lock	8	-2	0	1	4	8	-1	6	-2	-1	-2	0	2	-	-
c-MCS-TWA	61	-7	-1	4	20	61	-6	1	-3	-2	-1	-7	-1	2	-5
c-TTAS-MCS[30]	54	6	20	27	16	54	14	6	13	31	42	-	19	20	38
HCLH lock[40, 67]	30	-8	1	5	11	30	15	7	4	3	0	0	-1	-8	0
qspinlock[25]	24	-15	19	12	13	24	11	5	-6	-15	23	19	20	19	19

counter is modified by the waiting threads leads to an exponential explosion in the number of executions.

For completeness, we also tried other model checkers for WMMs. RMEM [81] could not finish any of the primitives using the binary code directly. It verified the semaphore implementation rewritten in pseudo-assembly in roughly two days. Surprisingly, Nidhugg [10] supports neither `atomic_cmpxchg` nor other atomic read-modify-write (RMW) operations on ARM and Power memory models—these operations are essential for any practical synchronization primitive. Finally, Power2SC [12] cannot translate our client code and crashes.

5.3 Optimizer Performance

We now evaluate the optimizer performance and the effectiveness of adaptive speculation. The optimizer employs AMC underneath. Table 2 (*Optimization time*) shows the time LR and ALR algorithms take to find a correct and maximally-relaxed barrier combination for each synchronization primitive. Except for already very fast cases (taking less than 1s), optimization time is greatly improved; this is especially noticeable for the qspinlock, which is reduced from 2.4 hours with LR to 11 minutes with ALR. In fact, the optimization time of ALR is dominated by the final verification: out of the total 657s for qspinlock, about 500s are spent for the final, non-speculative check (see AMC in *Verification time* column of Table 2).

5.4 Optimized Code Speedup

We now compare the performance of *sc* implementations of the synchronization primitives with their counterparts optimized with VSync. Our goal is to show the speedup a non-expert can see when implementing a primitive following the literature. In §6, we also compare expert-optimized implementations from open-source code.

5.4.1 *Microbenchmark.*

We start with a microbenchmark: each thread repeatedly acquires a (writer) lock, increments a shared counter, and releases the lock.

Setup. We experiment on a 128-core HiSilicon Kunpeng 920 with 2 sockets, 4 NUMA nodes (**ARMv8**), running Ubuntu 18.04 LTS – similar results can be produced in openEuler 20.09 [9]. We vary the number of threads from 1 to 127, run each experiment for a fixed period of time (30 seconds) and measure the throughput (critical sections per second). We run the experiments 5 times to ensure the stability of the results (for each case, we pick the median of these repeated runs). The benchmark runs as a Linux kernel module. We compare two variants of each primitive: an *sc*-only variant, and a VSync-optimized variant with barriers as shown in Table 2 (*Barriers after optimization*). We define speedup as $\frac{T_o}{T_s} - 1$, where T_o is median throughput of VSync-optimized and T_s is the median throughput of *sc*-only variants, respectively.

Results. Table 3 shows speedups for each synchronization primitive. The minimum (worst-case), the maximum (*best-case*), the mean and the median speedups are reported, together with detailed median speedups for each tested contention level (*i.e.*, number of threads). The throughput improved by up to 77.60% and on an average 8.57%. Perhaps surprisingly, in some isolated cases performance is not improved or even degrades (down to -40.57%). However, in 61.00% of cases spanning different levels of contention uniformly, speedup is positive and between 0.05%-77.60%. Our preliminary investigations reveal that degradation may occur due to a interplay between caches and exclusive accesses (*e.g.*, for `atomic_get_add`): if threads become too optimized, a thread currently holding a cache line can repeatedly invalidate exclusive accesses of other threads, resulting in starvation and performance degradation. Degradation may also occur due to speculative execution: with *sc*-barriers, the execution stops when a conditional branch cannot be resolved due

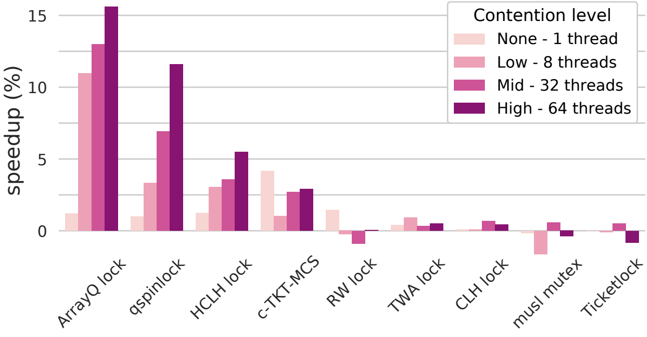


Figure 8: Speedup of VSync-optimized synchronization primitives on Kyoto Cabinet, varying contention level.

to a slow read operation, but without the barriers, the incorrect branch may be executed speculatively, leading to additional cache traffic. Finally, we observe that most of the best speedups are obtained with few threads. The benefits of barrier optimization tend to disappear when increasing the contention of the lock.

5.4.2 Real-World Workload. We evaluate the effect of barrier-mode optimization on Kyoto Cabinet [58], an in-memory key-value database. This database has a large synchronization overhead from a global reader-writer lock [29].

Setup. We replace `pthread_mutex` and `pthread_rwlock` with VSync-optimized and *sc*-only variants of our synchronization primitives from Table 2 using `LD_PRELOAD`. Experiments run a workload with 20% of writes for 30 seconds with different contention levels. We performed the experiments on the Kunpeng 920 server.

Results. Figure 8 shows the speedup of barrier relaxation for selected synchronization primitives at different contention levels. Overall, VSync-optimized variants outperform *sc*-only counterparts in most configurations, confirming the assumption that real-world applications are also affected by overconstrained barriers. This is most noticeable with the ArrayQ lock, which shows up to 17.5% improvement. Some of the locks such as musl mutex and CLH lock show no improvement with *no contention*. Only few barriers in the fast path of these primitives can be optimized. For Ticketlock, barrier optimization has an impact of less than 1%; for musl mutex and RW lock, less than 2%. Caching effects or thread execution noise seem to overshadow any optimization effect.

6 FINDINGS FROM USING VSYNC

6.1 Barrier Bug in seL4's CLH Lock

VSync detected a subtle bug in the (unverified) CLH big kernel lock of seL4. The maintainers confirmed the bug and merged our fix [8]. The relevant fragment of code is shown in Fig. 9 together with the execution order that leads to the bug. In this example, T_1 is about to enter the critical section and locks a node (`v = 1`) to block the next arriving thread. T_2 is next in line, waiting first for the node to be enqueued (by `q = &v`) and then for the node to be released again by T_1 (`v = 0`). However, in the execution order shown in the comments in Fig. 9, T_1 enqueues its node (step#1) before locking it (which only happens in step#4). T_2 sees the enqueued node (step#2) and enters the critical section (step#3), since it reads `*p = v = 0`. T_1 follows it into the critical section after locking its node (step#4), leading to a mutual exclusion violation. This is due to a missing *rel* barrier while enqueueing, which on ARM ensures that enqueueing (`q=&v`) becomes visible after the node is locked (`v=1`). We reproduced this bug on HiKey 960 and Kunpeng 920 platforms.

```
          v = 0, oldv = 0, q = &oldv, p = 0;

        T1                ||           T2
v = 1;   // step#4        ||  p = q;        // step#2
q = &v;  // step#1        ||  while
// CS                     ||    (*p == 1); // step#3
v = 0;   // step#5        ||  // CS
```

Figure 9: Simplified CLH code.

6.2 Challenges of MCS Locks on WMMs

MCS lock is a well-known algorithm, present in text books (*e.g.*, [40]), and widely used in the industry. We compare several implementations to illustrate the potentials of VSync-optimization, namely, providing similar performance as expert-optimization while guaranteeing the absence of barrier bugs.

Performance comparison. Figure 10 compares the performance of two expert-optimized MCS lock implementations, DPDK [36] and Concurrency Kit (ck) [16], with two VSync-optimized implementations, CertiKOS [39] and our own (from Table 2). The figure reports the throughput of the kernel-level microbenchmark (§5.4.1) on the Kunpeng 920 server.

CertiKOS MCS lock was implemented for x86 [5] and ported to WMM using VSync. Even with optimized barriers, its performance is worse than the other three – in some cases a third of the performance – because it only implements the MCS lock slowpath case, which simplifies their verification but incurs severe performance degradation. DPDK and ck implementations are rather similar and mostly employ *fences* (*i.e.*, explicit barriers), which is a common technique for x86, but can be expensive on ARMv8 [62]. Our MCS lock implementation follows the same implementation pattern but

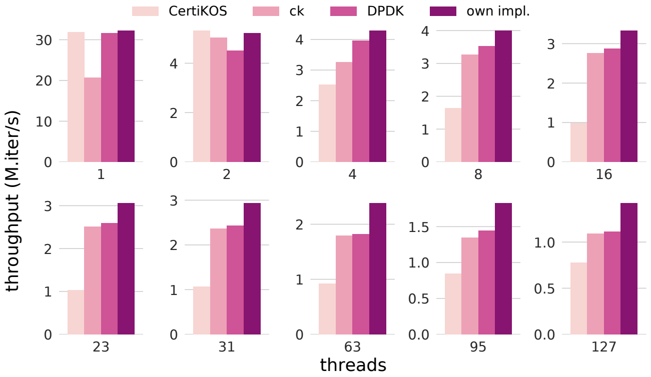


Figure 10: Throughput of MCS lock implementations varying number of threads with kernel-level microbenchmark.

```
 1 rte_mcslock_lock(rte_mcslock_t **msl, rte_mcslock_t *me) {
 2   __atomic_store_n(&me->locked, 1, __ATOMIC_RELAXED);
 3   __atomic_store_n(&me->next, NULL, __ATOMIC_RELAXED);
 4   prev = __atomic_exchange_n(msl, me, __ATOMIC_SEQ_CST);
 5   if (prev == NULL)
 6     return;
 7   __atomic_store_n(&prev->next, me, __ATOMIC_RELAXED);
 8   __atomic_thread_fence(__ATOMIC_SEQ_CST);
 9   while (__atomic_load_n(&me->locked, __ATOMIC_ACQUIRE));
10 }
11
12 rte_mcslock_unlock(rte_mcslock_t **msl, rte_mcslock_t *me) {
13   if (__atomic_load_n(&me->next, __ATOMIC_RELAXED) == NULL) {
14     // **ignore this branch**
15   }
16   /* Pass lock to next waiter. */
17   __atomic_store_n(&me->next->locked, 0, __ATOMIC_RELEASE);
18 }
```

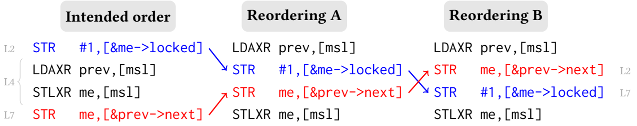
Figure 11: Simplified DPDK MCS lock with bug. The write in Line 7 should have an `__ATOMIC_RELEASE` barrier instead.

relies on implicit barriers; it has similar performance as the expert-optimized ones – sometimes even higher. The difference can be partially attributed to the barrier selection, but other aspects such as data and code cacheline alignment also play an important role. One can only expect VSync-optimized implementations to perform similarly as expert-optimized ones if other implementation factors do not hamper the performance.

Bugs. We also found bugs in two MCS implementations: First, a bug causing mutual exclusion violations was introduced in the MCS lock implementation of an internal Huawei product while porting it from x86 to ARM. The engineers had added several barriers to the code trying to enforce the right ordering of instructions. Nevertheless, VSync identified a missing *acq* barrier before returning from the lock's acquire function. Moreover, VSync identified two of the barriers introduced by the engineers as redundant.

Second, in the expert-optimized DPDK MCS lock, VSync uncovered a scenario similar to Fig. 1, in which T_1 fails to acquire the lock and T_2 is about to release the lock. Due to a misplaced barrier, T_1 may miss T_2's update of the locked field and spin forever waiting for locked to become 0. Consider the simplified code in Fig. 11. T_1 enters rte_mcslock_lock, initializes locked to 1 (Line 2) and exchanges the tail of the queue (Line 4). Afterwards, T_1 sets prev->next to its node (Line 7). Note that prev points to T_2's node since T_2 holds the lock. Next, T_2 enters rte_mcslock_unlock, reads me->next pointing to T_1's node and sets locked to 0 (Line 17). Unfortunately, the write of Line 17 may be overwritten by Line 2, causing T_1 to hang at Line 9. The lost update can happen because Lines 2 and 7 can be reordered in spite of the exchange between them having an *sc* barrier (Line 4).

Figure 12 illustrates how this reorder occurs in pseudo-assembly. The exchange operation (as well as any read-modify-write operation) can be implemented with load-exclusive/store-exclusive. Since the exchange operation has an *sc* barrier, its implementation has an *acq* barrier in the load-exclusive (the A in LDAXR) and a *rel* barrier in the store-exclusive (the L in STLXR). The reordering **A** is possible because the *acq* barrier does not stop preceding stores from taking effect after it, and the *rel* barrier does not stop succeeding stores



```
       Intended order               Reordering A                 Reordering B

L2   STR   #1,[&me->locked]    LDAXR prev,[msl]            LDAXR prev,[msl]
   { LDAXR prev,[msl]          STR   #1,[&me->locked]      STR   me,[&prev->next]   L2
L4 { STLXR me,[msl]            STR   me,[&prev->next]      STR   #1,[&me->locked]   L7
L7   STR   me,[&prev->next]    STLXR me,[msl]              STLXR me,[msl]
```

Figure 12: Bug manifestation in DPDK MCS lock (pseudo-assembly for ARMv8).

from taking effect before it. The reordering **B** is possible because both stores have *rlx* mode.

VSync suggests to add a *rel* barrier to the write of Line 7, which makes reordering **B** impossible. The DPDK maintainers confirmed the bug and merged our patch [4].

6.3 Applying VSync to Linux's qspinlock

Over the course of over two years experts repeatedly optimized barriers in the qspinlock [26, 63, 90] culminating in version 5.6 [65]. We rolled back these optimizations to the partially optimized version 4.4 [64]. In order to isolate the performance impact of barriers, we applied all optimizations unrelated to barriers from 5.6 to 4.4, then we ported it to VSync. VSync recommended barrier modes similar to those used by the experts (see Table 4) in roughly 11 minutes – in contrast, the expert optimization took several release cycles. We cannot verify that 5.6 is correct because currently VSync does not support LKMM, the memory model used in Linux. We expect that VSync extended with an LKMM module would produce the same barriers as in 5.6 – we leave such extensions as future work. Figure 13 compares the throughput of version 4.4 of qspinlock, the one optimized by experts (5.6) and one automatically optimized. VSync-optimized qspinlock performs up to 9% faster than version 4.4 and on par with version 5.6 (less than 1% difference).

6.4 Optimizing musl's Mutual Exclusion Lock

We next considered musl [7], a fast and lightweight C library. We applied VSync to the pthread_mutex implementation of the latest musl version (1.2.0). The implementation is overly cautious: it contains nine barriers in *sc*-mode sprayed around the atomic increments and CAS operations. VSync optimizes the nine *sc* barriers into two *acq* and one *rel*, fully relaxing the remainder six. Notwithstanding, Fig. 13 shows negligible performance differences when running Kyoto Cabinet. The time needed to park and awake threads in the kernel hides any optimization gains.

Specifically, the musl mutex spins for a fixed number of iterations in userspace; if the lock does not become available in that time, the mutex traps into the kernel. The optimized barriers can

Table 4: Barrier optimization results for Linux's qspinlock.

Version	*acq*	*rel*	*sc*	Time	Correctness
Linux 4.4[64]	3	6	6	2015/09/11	Not verified
Linux 4.5[63]	6	2	1	2015/11/09	Barrier bug, fixed in [26]
Linux 4.8[90]	6	3	0	2016/06/03	Barrier bug, fixed in [26]
Linux 4.16[26]	6	4	0	2018/02/13	Not verified
Linux 5.6[65]	6	2	1	2020/01/07	Not verified
VSync	7	2	1	11 minutes	VSync-verified

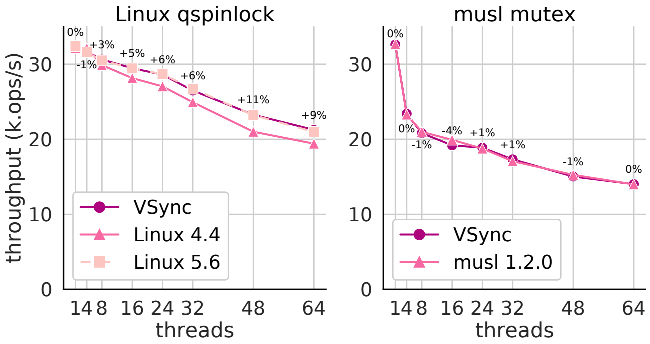


Figure 13: Kyoto Cabinet with different versions of Linux qspinlock and musl mutex. The annotation over the points represent the speedup of VSync-optimized variants compared to Linux 4.4 and musl 1.2.0 versions, respectively.

cause the fixed number of spin loop iterations to pass more quickly. Consequently, the optimized mutex traps into the kernel earlier and misses the lock more frequently, and throughput degrades. This shows that optimizing barriers does not always provide the expected improvements for the application. For the sake of experiment, we also compared the performance without thread parking (mutexes stay and spin in usermode); in this comparison, the optimization slightly improved performance on the Kyoto Cabinet by 1-3% for all thread number configurations.

6.5 Bug in NUMA-Aware Readers-Writer Lock

We conclude our findings with a bug that VSync detected in the NUMA-aware readers-writer lock from Kashyap *et al* [49]. Their implementation fails to provide exclusivity for writers on ARM because the lock function of readers uses only *rlx* operations (see Line 222 of `cmcsmcsrw.c` [84]). This issue is another example of the difficulty of porting from x86 to ARM. VSync helps with such porting by hiding this complexity and returning verified and portable code with high performance.

7 FURTHER RELATED WORK

Verification of systems software. Many formal verification efforts such as Hyperkernel [73], Serval [72], or CVM [78] consider only non-concurrent software, but can be easily ported to multiprocessors with WMMs if correct big locks were provided which allow sequential reasoning in the critical sections. Similarly, verified systems software like the seL4 microkernel [51] and concurrent file systems [22, 92] exclude synchronization primitives as external parts to be verified independently. If this is not done, it leaves a verification gap leading potentially to bugs in a formally verified system, such as the mutual exclusion violation due to the overly-relaxed barrier in the seL4 big kernel lock [79]. VSync closes this gap, and thus complements the verification of these systems.

Other formal verification projects such as CertiKOS [39], Verisoft XT [19], and Armada [66] have also formally verified synchronization primitives, either by manual Coq proofs [50] or by automatic formal verifiers [41, 66] with some manual assistance. These verification efforts are considerably more general as they verify the correctness of the locks under arbitrary client code, while we verify the locks with a specific client. This can potentially cause issues if the specific client is not general enough, *e.g.*, has insufficiently many threads. In contrast to our work, the previously mentioned verification took considerable annotation and proof effort, making it infeasible for today's industrial environments. Our effort is fully automatic; verifying, *e.g.*, the qspinlock takes 11 minutes of wall-clock time and little human effort. Furthermore, existing work does not consider WMMs beyond x86, resulting in suboptimal barriers.

Table 5: Model checking for concurrent programs.

Model Checking Technique	SeqCst		WMM	
	safety	AT	safety	AT
Stateful [20, 42, 60, 61, 69, 81, 91]	✓	✓	*not scalable*	
Stateless [10, 13, 14, 27, 43, 52, 54–56, 71]	✓	✗	✓	✗
Our AMC	✓	✓	✓	✓

Model checking concurrent programs. A summary of existing model checkers is given in Table 5. Stateful model checkers can tackle programs with awaits but do not scale to WMMs due to state space explosion. For example, they store every possible order in which operations execute and propagate between cores. Initially, we attempted to verify synchronization primitives with RMEM [81], a stateful model checker for WMMs. Even for simple locks, verification took multiple days. We then switched to the stateless model checker GenMC [54, 56], which with appropriately chosen loop unroll bounds verifies the same locks within seconds. As mentioned before, stateless model checkers cannot decide termination, resulting in hangs of "verified" locks on real hardware. AMC solves this and can decide termination of awaits while mitigating the state space explosion problem.

Automatic barrier optimization. The automatic optimization (or placement) of barriers is a challenging and promising area of research. Prior efforts [17, 24, 32, 76, 88] focus on the barriers necessary to guarantee a stronger memory model, *e.g.*, to ensure that the program has only sequentially consistent behaviors. In contrast, we deduce the barriers necessary to make the code correct. Besides supporting implicit barriers, the advantage of our approach is that it allows much more relaxed barriers, *e.g.*, if relaxing some barrier introduces non-sequentially consistent behaviors that do not violate mutual exclusion or await termination.

Notwithstanding, model checking approaches such as ours may profit from so called robustness theorems (*e.g.*, [24]), which state that certain barrier placements are sufficient for preserving the stronger memory model; these theorems sometimes suggest thread-local conservative heuristics for barrier placement. While our work requires the complete analysis of the possible interactions of threads and is therefore not scalable to large software systems, such thread-local heuristics, or using static over-approximation of program executions [17], are scalable to large code bases, at the cost of introducing more barriers than necessary to preserve the stronger memory model (*e.g.*, [76]).

8 CONCLUSION AND FUTURE WORK

Optimization of synchronization primitives on WMMs is not a task that should be left to humans. Despite their considerable effort, experts miss optimization opportunities and sometimes even introduce subtle bugs. This is evidenced by the three barrier bugs in code optimized by experts for WMMs.

The automated approach of VSync produces efficient barriers and no bugs. To achieve this, we developed a method to detect nonterminating awaits that can be adopted by stateless model checkers. This method can be applied to more than just synchronization primitives. As future work, we would like to apply VSync to the optimization and verification of lock-free and other concurrent data structures.

ACKNOWLEDGMENTS

We sincerely thank our shepherd Caroline Trippel and our anonymous reviewers for their insightful comments and suggestions. We also thank Jitang Lei for suggesting the MCS lock in the internal Huawei product and for supporting us throughout the project. Finally, we thank our colleagues of Huawei DRC and OS Kernel lab for reviewing the draft of this work.

A ARTIFACT APPENDIX

A.1 Abstract

VSync is a framework that allows users to verify and optimize synchronization primitives on WMM (such as ARMv8). It consists of a model checker (AMC), an optimizer (ALR) and a library of atomic operations (VSync-atomics). The **main results** of the paper are the detection of bugs in existing implementations of synchronization primitives (§6) and the verification and optimization of several synchronization primitives (Table 2), including the performance of the model checker (verification time of GenMC versus AMC) and of the optimizer (optimization time of ALR versus LR). The secondary results are the **performance experiments** comparing several synchronization primitives optimized (and not optimized) with VSync. The performance experiments include Table 3 as well as Figures 8, 10 and 13. The artifact archive groups the benchmarks according to these two sets of evaluations: *main results* and *performance experiments*.

All the evaluations run preferably on Ubuntu 18.04. The *main results* can be reproduced on any `x86_64` workstation with 16 GB of RAM and 4 or more cores. The *performance experiments* require a large `ARMv8` server. Our experiments ran on a Kunpeng 920 processor with 4 NUMA nodes and 128 cores, which is made remotely available to the artifact reviewers.

We expect the artifact evaluation to reproduce the following *main results*: (1) the detection of 3 bugs in existing synchronization primitives; (2) the detection of await termination (AT) violations in half of the evaluated synchronization primitives; and (3) ALR optimization being significantly faster than LR for most synchronization primitives. Moreover, we expect the *performance experiments* to show that VSync-optimized synchronization primitives perform similarly to expert-optimized ones, in particular, for Linux qspinlock and musl mutex.

Required knowledge to run the artifacts are:

- Basic Linux command line (installing packages, running and slightly modifying scripts);
- Building Docker images and running Docker containers;
- Configuring SSH hosts to access our internal server;
- Know how to change Linux kernel boot arguments.

A.2 Artifact Checklist

- **Algorithm:** Model checking, optimization, and synchronization primitives (*e.g.*, MCS lock).
- **Program:** Model checker, optimizer, kernel module, pthread interposition library, and benchmarks (*e.g.*, Kyoto Cabinet `v1.2.76`) are all included in the archive as source code.
- **Compilation:** Clang, LLVM, GCC, and other build tools available in Ubuntu 18.04 packages. Golang 1.13 used from Docker image.
- **Transformations:** The source code of the barrier mutator is in the archive. It is built in a Docker image (Dockerfile is also included).
- **Binary:** no binary included, only source files.
- **Run-time environment:** Ubuntu 18.04, root access and Docker.
- **Hardware:** The *main results* can be reproduced on any `x86_64` workstation with 16 GB RAM and 4 or more cores. The *performance experiments* require a large `ARMv8` server. Our experiments ran on a Kunpeng 920 processor with 4 NUMA nodes and 128 cores. The server is available to the artifact reviewers on demand.
- **Run-time state:** The performance experiments are sensitive to runtime state. To reduce variability, scripts (requiring root access) are provided to ensure core isolation and fix the CPU frequency.
- **Execution:** To avoid interference, we strongly recommend not running any other tasks simultaneously. Avoid running multiple connections to the server with multiple users. Other precautions (such as process pinning) are covered by the provided scripts. The runtime to reproduce the paper results is between 2 and 3 days, but the parameters can be tuned to reduce that duration.
- **Metrics:** The *main results* report AT-detection requirement (as a Boolean) as well as verification and optimization times (in seconds). The *performance experiments* report absolute throughput of different benchmarks (iterations per second) or relative speedups.
- **Output:** The evaluations output an `HTML` file containing a rendered Jupyter Notebook. The notebook includes the tables and figures of the paper, reproduced with the selected dataset (either at the time of submission, or freshly-generated datasets). Generated results include Table 2, Table 3, Figure 8, Figure 10 and Figure 13.
- **Experiments:** The steps to reproduce are documented in the `README` files inside the archive. Every step is automated, and changing parameters only requires slight modification in shell scripts. See §A.6 for the maximum expected variation of results.
- **How much disk space required? :** Not more than 10 GiB.
- **How much time is needed to prepare workflow? :** Approximately 2 to 3 hours to setup the machine, install the dependencies, build the Docker image and run the scripts.
- **How much time is needed to complete experiments (approximately)? :** The *main results* take 48 to 72 hours to fully run, but a smaller set of synchronization primitives can be selected. The *performance experiments* take 48 hours to fully run, but shorter durations and/or fewer repetitions can be configured.
- **Publicly available? :** Except for the dependencies, the source code in the archive is **closed source**. We are working towards open sourcing the library and tools in order to make the artifact publicly accessible in the future.
- **Workflow framework used? :** We use custom scripts and makefiles to automate the generation of the results.

A.3 Description

A.3.1 How to Access. The artifacts for this submission are distributed to the reviewers as a password-protected zip file that can be downloaded from Dropbox: *undisclosed.*

The `README.md` file in the root of the archive details the preparation and execution of the *main results* and *performance experiments.* The `README.html` contains the same information in HTML format, with links to other files in the archive.

A.3.2 Hardware Dependencies. The *main results* were performed on a workstation with a 6-core (12-hyperthreads) Intel Xeon W-2133 at 3.60GHz and 16GB RAM.

The *performance experiments* were performed in the following platform: a 128-core HiSilicon Kunpeng 920 with 2 sockets, 4 NUMA nodes. We offered the artifact reviewers remote access to the machine we used for the evaluations of our paper.

A.3.3 Software Dependencies. The dependencies are essentially packages that can be installed with `apt` on Ubuntu 18.04: clang, LLVM, GCC, and other build tools. Docker has to be installed by following instructions on its website. The `README` files inside the archive describe how to install all the dependencies.

A.3.4 Data Sets.

- **`00-submission`:** The dataset used to generate the tables and figures of the first submission of the paper.
- **`01-camera-ready`:** A fresh run of experiments ran by ourselves before submitting the artifacts. We used this dataset to generate the tables and figures of the camera-ready version of the paper.
- **`02-artifact-reviewer`:** An empty dataset for use by the reviewer. By default, the benchmark scripts will output the generated results in that directory.

A.4 Installation

The *main results* require the installation of Docker and `make` and the subsequent creation of a Docker image by simply running `make`. The *performance experiments* require mainly GCC, `make` and other build essentials. For generating the plots, Python 3 and `venv` are necessary. Please, see the `README` files inside the archive for detailed instructions on how to prepare the artifacts and benchmarks.

A.5 Experiment Workflow

Please, see `README` files inside the archive for detailed instructions on how to execute the evaluations.

Here is a high-level workflow.

1. Generate notebook with `00-submission` dataset.
2. On `x86_64` machine: reproduce *main results.*
3. On `ARMv8` server: reproduce *performance experiments.*
4. Copy the results (`CSV` and `DAT` files) of steps 2 to 4 into the `02-artifact-reviewer` dataset on the same machine.
5. Generate notebook with `02-artifact-reviewer` dataset.
6. Generate notebook with `01-camera-ready` dataset.
7. Compare the notebooks generated in steps 1, 5 and 6.

A.6 Evaluation and Expected Results

A.6.1 Expected Results. We expect the artifact evaluation to reproduce the following *main results*:

- the detection of 3 bugs in existing synchronization primitives (seL4 CLH, DPDK MCS, Huawei MCS locks);
- the detection of await termination (AT) violations in half of the evaluated synchronization primitives when barriers are too weak ("Requires AT detection" column in Table 2); and
- ALR optimization being significantly faster than LR for most synchronization primitives (from 40% to 95% faster for optimizations taking longer than 1 second).

Moreover, we expect the reproduction of the key result of the *performance experiments*: VSync-optimized synchronization primitives perform similarly to expert-optimized ones, in particular, for Linux qspinlock (around 1% difference to version 5.6) and musl mutex (less than 4%).

The bug results can be inspected in the generated logs as described in the `README` files. All other results can be inspected in the final HTML-rendered Jupyter Notebook.

A.6.2 Expected Variations. In the *main results*, we expect no difference in bug and AT-violation detection; and no significant variation in the *relation* of ALR and LR optimization times.

The *performance experiments* are highly sensitive to the run-time and machine specification; variations will be observed. To support the paper claims, it is sufficient to observe that for most configurations, VSync-optimized primitives are (1) more performant than SC-only and (2) perform similarly to expert-optimized ones.

A.7 Experiment Customization

The *main results* can be configured to run a smaller set of synchronization primitives. The *performance experiments* can be configured with different duration, number of repetitions, number of cores and NUMA topology (according to the reviewer's machine). The interested reviewer can implement, verify and optimize new synchronization primitives by following the VSync tutorial. Please, see `README` files inside the archive for detailed instructions. Such primitive can then be evaluated using the scripts and programs provided for the *performance experiments.*

A.8 Notes

Since the submission of the paper, there has been an important bug fixed in the underlying GenMC v0.5.3; we backported it to AMC. This bugfix results in slightly different verification and optimization times and a more relaxed barrier combinations in the cohort synchronization primitives, *i.e.*, c-MCS-TWA, c-TTAS-MCS and c-TKT-MCS. See `01-camera-ready` dataset for comparison.

REFERENCES

[1] 1999. spin_unlock optimization(i386). https://marc.info/?l=linux-kernel&m=94318921016232&w=2.

[2] 2008. Linux Ticketlock. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=314cdbefd1fd0a7acf3780e9628465b77ea6a836.

[3] 2018. Linux-Kernel Memory Model. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0124r6.html.

[4] 2020. Await termination violation bug fix in DPDK. http://patches.dpdk.org/patch/75983/.

[5] 2020. CetiKOS MCS lock implementation – source code. https://certikos.github.io/certikos-artifact/html/mcertikos.mcslock.MMCSLockAbsIntroCSource.html.

[6] 2020. Clang: C Language Family Frontend for LLVM. https://clang.org.

[7] 2020. musl libc: an implementation of the C standard library. https://musl.libc.org.

[8] 2020. Mutual exclusion bug fix in seL4. https://github.com/seL4/seL4/pull/199/commits.

[9] 2020. openEuler. https://openeuler.org.

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

[11] Parosh Aziz Abdulla, Jatin Arora, Mohamed Faouzi Atig, and Shankaranarayanan Krishna. 2019. Verification of Programs under the Release-Acquire Semantics. 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, 1117–1132. https://doi.org/10.1145/3314221.3314649

[12] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. 2017. Context-Bounded Analysis for POWER. In *Proceedings, Part II, of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 10206*. Springer-Verlag, Berlin, Heidelberg, 56–74. https://doi.org/10.1007/978-3-662-54580-5_4

[13] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. 2016. Stateless model checking for POWER. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 134–156.

[14] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. 2018. Optimal Stateless Model Checking under the Release-Acquire Semantics. *Proceedings of the ACM on Programming Languages* 2, OOPSLA, Article 135 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276505

[15] Sarita V. Adve and Mark D. Hill. 1990. Weak Ordering—a New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* (Seattle, Washington, USA) *(ISCA '90)*. Association for Computing Machinery, New York, NY, USA, 2–14. https://doi.org/10.1145/325164.325100

[16] Samy Al Bahra. 2015. Concurrency kit. *Retrieved November* 8 (2015), 2018. https://github.com/concurrencykit/ck.

[17] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. 2014. Don't Sit on the Fence. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 508–524.

[18] 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

[19] Bernhard Beckert and Michał Moskal. 2010. Deductive Verification of System Software in the Verisoft XT Project. *KI - Künstliche Intelligenz* 24, 1 (2010), 57–61. https://doi.org/10.1007/s13218-010-0005-7

[20] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. 2007. The Software Model Checker Blast: Applications to Software Engineering. *International Journal on Software Tools for Technology Transfer (STTT)* 9, 5-6 (2007), 505–525. https://doi.org/10.1007/s10009-007-0044-z

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

[22] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying Concurrent, Crash-Safe Systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 243–258. https://doi.org/10.1145/3341301.3359632

[23] Soham Chakraborty and Viktor Vafeiadis. 2016. Validating optimizations of concurrent C/C++ programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. 216–226.

[24] Ernie Cohen and Norbert Schirmer. 2009. A Better Reduction Theorem for Store Buffers. *CoRR* abs/0909.4637 (2009). arXiv:0909.4637 http://arxiv.org/abs/0909.4637

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

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

[27] Brian Demsky and Patrick Lam. 2015. SATCheck: SAT-directed stateless model checking for SC and TSO. In *OOPSLA 2015*. ACM, New York, NY, USA, 20–36. https://doi.org/10.1145/2814270.2814297

[28] Dave Dice and Alex Kogan. 2019. TWA – Ticket Locks Augmented with a Waiting Array. In *European Conference on Parallel Processing*. Springer, 334–345.

[29] Dave Dice, Alex Kogan, Yossi Lev, Timothy Merrifield, and Mark Moir. 2014. Adaptive Integration of Hardware and Software Lock Elision Techniques. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures* (Prague, Czech Republic) *(SPAA '14)*. Association for Computing Machinery, New York, NY, USA, 188–197. https://doi.org/10.1145/2612669.2612696

[30] David Dice, Virendra J. Marathe, and Nir Shavit. 2015. Lock Cohorting: A General Technique for Designing NUMA Locks. *ACM Trans. Parallel Comput.* 1, 2, Article 13 (Feb. 2015), 42 pages. https://doi.org/10.1145/2686884

[31] Ulrich Drepper. 2005. Futexes are tricky. *Futexes are Tricky, Red Hat Inc, Japan* 4 (2005).

[32] Xing Fang, Jaejin Lee, and Samuel P. Midkiff. 2003. Automatic Fence Insertion for Shared Memory Multiprocessing. In *Proceedings of the 17th Annual International Conference on Supercomputing* (San Francisco, CA, USA) *(ICS '03)*. Association for Computing Machinery, New York, NY, USA, 285–294. https://doi.org/10.1145/782814.782854

[33] C. Flanagan, S. N. Freund, and S. Qadeer. 2005. Exploiting purity for atomicity. *IEEE Transactions on Software Engineering* 31, 4 (2005), 275–291.

[34] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005*. ACM, 110–121. https://doi.org/10.1145/1040305.1040315

[35] 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 *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 608–621. https://doi.org/10.1145/2837614.2837615

[36] Linux Foundation. 2015. Data Plane Development Kit (DPDK). http://www.dpdk.org

[37] Natalia Gavrilenko, Hernán Ponce-de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2019. BMC for Weak Memory Models: Relation Analysis for Compact SMT Encodings. In *International Conference on Computer Aided Verification*. Springer, 355–365.

[38] Ronghui Gu, Zhong Shao, Hao Chen, Jieung Kim, Jérémie Koenig, Xiongnan (Newman) Wu, Vilhelm Sjöberg, and David Costanzo. 2019. Building Certified Concurrent OS Kernels. *Commun. ACM* 62, 10 (Sept. 2019), 89–99. https://doi.org/10.1145/3356903

[39] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 653–669.

[40] Maurice Herlihy and Nir Shavit. 2011. *The art of multiprocessor programming*. Morgan Kaufmann.

[41] Mark A Hillebrand and Dirk C Leinenbach. 2009. Formal verification of a reader-writer lock implementation in C. *Electronic Notes in Theoretical Computer Science* 254 (2009), 123–141.

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

[43] Alan Huang. 2016. Maximally Stateless Model Checking for Concurrent Bugs under Relaxed Memory Models. In *Proceedings of the 38th International Conference on Software Engineering Companion* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 686–688. https://doi.org/10.1145/2889160.2891042

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

[45] ISO/IEC. 2011. Committee Draft N1570 of C11 standard.

[46] Ranjit Jhala and Rupak Majumdar. 2009. Software Model Checking. *ACM Comput. Surv.* 41, 4, Article 21 (Oct. 2009), 54 pages. https://doi.org/10.1145/1592434.1592438

[47] Bengt Jonsson. 2009. State-Space Exploration for Concurrent Algorithms under Weak Memory Orderings: (Preliminary Version). *SIGARCH Comput. Archit. News* 36, 5 (June 2009), 65–71. https://doi.org/10.1145/1556444.1556453

[48] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-Memory Concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 175–189. https://doi.org/10.1145/3009837.3009850

[49] Sanidhya Kashyap, Irina Calciu, Xiaohe Cheng, Changwoo Min, and Taesoo Kim. 2019. Scalable and Practical Locking with Shuffling. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 586–599. https://doi.org/10.1145/3341301.3359629

[50] Jieung Kim, Vilhelm Sjöberg, Ronghui Gu, and Zhong Shao. 2017. Safety and liveness of MCS lock—Layer by layer. In *Asian Symposium on Programming Languages and Systems*. Springer, 273–297.

[51] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[52] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2017. Effective Stateless Model Checking for C/C++ Concurrency. *Proceedings of the ACM on Programming Languages* 2, POPL, Article 17 (Dec. 2017), 32 pages. https://doi.org/10.1145/3158105

[53] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Effective Lock Handling in Stateless Model Checking. *Proceedings of the ACM on Programming Languages* 3, OOPSLA, Article 173 (Oct. 2019), 26 pages. https://doi.org/10.1145/3360599

[54] 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

[55] Michalis Kokologiannakis and Konstantinos Sagonas. 2017. 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* (Santa Barbara, CA, USA) *(SPIN 2017)*. Association for Computing Machinery, New York, NY, USA, 172–181. https://doi.org/10.1145/3092282.3092287

[56] Michalis Kokologiannakis and Viktor Vafeiadis. 2020. HMC: Model Checking for Hardware Memory Models. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1157–1171. https://doi.org/10.1145/3373376.3378480

[57] Michael Kuperstein, Martin Vechev, and Eran Yahav. 2012. Automatic Inference of Memory Fences. *SIGACT News* 43, 2 (June 2012), 108–123. https://doi.org/10.1145/2261417.2261438

[58] FAL Labs. 2011. Kyoto Cabinet: A straightforward implementation of DBM. http://fallabs.com/kyotocabinet.

[59] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 618–632. https://doi.org/10.1145/3062341.3062352

[60] Kim G Larsen, Paul Pettersson, and Wang Yi. 1997. UPPAAL in a nutshell. *International journal on software tools for technology transfer* 1, 1-2 (1997), 134–152.

[61] Stella Lau, Victor BF Gomes, Kayvan Memarian, Jean Pichon-Pharabod, and Peter Sewell. 2019. Cerberus-BMC: A Principled Reference Semantics and Exploration Tool for Concurrent and Sequential C. In *International Conference on Computer Aided Verification*. Springer, 387–397.

[62] Nian Liu, Binyu Zang, and Haibo Chen. 2020. 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* (San Diego, California) *(PPoPP '20)*. Association for Computing Machinery, New York, NY, USA, 348–361. https://doi.org/10.1145/3332466.3374535

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

[64] Waiman Long and Peter Zijlstra. 2015. 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.

[65] Waiman Long and Peter Zijlstra. 2020. 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.

[66] Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. 2020. Armada: Low-Effort Verification of High-Performance Concurrent Programs. 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, 197–210. https://doi.org/10.1145/3385412.3385971

[67] Victor Luchangco, Dan Nussbaum, and Nir Shavit. 2006. A hierarchical CLH queue lock. In *European Conference on Parallel Processing*. Springer, 801–810.

[68] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. 2012. An Axiomatic Memory Model for POWER Multiprocessors. In *Computer Aided Verification*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 495–512.

[69] Kenneth L McMillan. 1993. Symbolic model checking. In *Symbolic Model Checking*. Springer, 25–60.

[70] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (Feb. 1991), 21–65. https://doi.org/10.1145/103727.103729

[71] Madanlal Musuvathi and Shaz Qadeer. 2008. Fair Stateless Model Checking. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 362–371. https://doi.org/10.1145/1375581.1375625

[72] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 225–242. https://doi.org/10.1145/3341301.3359641

[73] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 252–269. https://doi.org/10.1145/3132747.3132748

[74] Tuan-Phong Ngo. 2019. *Model Checking of Software Systems under Weak Memory Models*. Ph.D. Dissertation. Acta Universitatis Upsaliensis.

[75] Brian Norris and Brian Demsky. 2016. A Practical Approach for Model Checking C/C++11 Code. *ACM Trans. Program. Lang. Syst.* 38, 3, Article 10 (May 2016), 51 pages. https://doi.org/10.1145/2806886

[76] Jonas Oberhauser. 2015. A Simpler Reduction Theorem for X86-TSO. In *Revised Selected Papers of the 7th International Conference on Verified Software: Theories, Tools, and Experiments - Volume 9593* (San Francisco, CA, USA) *(VSTTE 2015)*. Springer-Verlag, Berlin, Heidelberg, 142–164. https://doi.org/10.1007/978-3-319-29613-5_9

[77] Jonas Oberhauser, Rafael Lourenco de Lima Chehab, Diogo Behrens, Ming Fu, Antonio Paolillo, Lilith Oberhauser, Koustubha Bhat, Yuzhong Wen, Haibo Chen, Jaeho Kim, and Viktor Vafeiadis. 2021. VSync: Push-Button Verification and Optimization for Synchronization Primitives on Weak Memory Models (Technical Report). arXiv:2102.06590 [cs.LO]

[78] Wolfgang J. Paul, Christoph Baumann, Petro Lutsyk, and Sabine Schmaltz. 2016. *System Architecture - An Ordinary Engineering Discipline*. Springer. https://doi.org/10.1007/978-3-319-43065-2

[79] Sean Peters, Adrian Danis, Kevin Elphinstone, and Gernot Heiser. 2015. For a Microkernel, a Big Lock Is Fine. In *Proceedings of the 6th Asia-Pacific Workshop on Systems* (Tokyo, Japan) *(APSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 3, 7 pages. https://doi.org/10.1145/2797022.2797042

[80] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the Gap between Programming Languages and Hardware Weak Memory Models. *Proceedings of the ACM on Programming Languages* 3, POPL, Article 69 (Jan. 2019), 31 pages. https://doi.org/10.1145/3290382

[81] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2017. Simplifying ARM Concurrency: Multicopy-Atomic Axiomatic and Operational Models for ARMv8. *Proceedings of the ACM on Programming Languages* 2, POPL, Article 19 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158107

[82] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. X86-TSO: A Rigorous and Usable Programmer's Model for X86 Multiprocessors. *Commun. ACM* 53, 7 (July 2010), 89–97. https://doi.org/10.1145/1785414.1785443

[83] SPARC International Inc. 1994. *The SPARC architecture manual (version 9)*. Prentice-Hall.

[84] Georgia Tech SS Lab. 2010. NUMA-aware Reader-Writer Implementation. https://github.com/sslab-gatech/shfllock/blob/master/benchmarks/kernel-syncstress/locks/cmcsmcsrw.c#L222.

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

[86] Viktor Vafeiadis. 2017. Program verification under weak memory consistency using separation logic. In *International Conference on Computer Aided Verification*. Springer, 30–46.

[87] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. 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* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 209–220. https://doi.org/10.1145/2676726.2676995

[88] Viktor Vafeiadis and Francesco Zappa Nardelli. 2011. Verifying Fence Elimination Optimisations. In *Static Analysis*, Eran Yahav (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 146–162.

[89] Andrew Waterman and Krste Asanović (Eds.). 2019. The RISC-V Instruction Set Manual. https://content.riscv.org/wp-content/uploads/2019/06/riscv-spec.pdf. Accessed: 2020-03-06.

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

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

[92] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. 2019. Using Concurrent Relational Logic with Helpers for Verifying the AtomFS File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 259–274. https://doi.org/10.1145/3341301.3359644