Verifying and Optimizing Compact NUMA-Aware Locks on Weak Memory Models

Antonio Paolillo[1] Hernán Ponce de León[2] Thomas Haas[3] Diogo Behrens[1]
Rafael Chehab[1] Ming Fu[1] Roland Meyer[3]

[1]Huawei Dresden Research Center, Germany
[2]University of the Bundeswehr Munich, Germany
[3]TU Braunschweig, Germany

Abstract

Developing concurrent software is challenging, especially if it has to run on modern architectures with Weak Memory Models (WMMs) such as ARMv8, Power, or RISC-V. For the sake of performance, WMMs allow hardware and compilers to aggressively reorder memory accesses. To guarantee correctness, developers have to carefully place memory barriers in the code to enforce ordering among critical memory operations.

While WMM architectures are growing in popularity, identifying the necessary and sufficient barriers of complex synchronization primitives is notoriously difficult. Unfortunately, publications often consider barriers to be just implementation details and omit them. In this technical note, we report our efforts in verifying the correctness of the Compact NUMA-Aware (CNA) lock algorithm on WMMs. The CNA lock is of special interest because it has been proposed as a new slowpath for Linux `qspinlock`, the main spinlock in Linux.

Besides determining a correct and efficient set of barriers for the original CNA algorithm on WMMs, we investigate the correctness of Linux `qspinlock` and the latest Linux CNA patch (v15) on the memory models LKMM, ARMv8, and Power. Surprisingly, we have found that Linux `qspinlock` and, consequently, Linux CNA are incorrect according to LKMM, but are still correct when compiled to ARMv8 or Power.

1 Introduction

Correctly developing concurrent software is challenging because of the gigantic number of interleavings of thread executions. Developing such software to run on modern architectures with Weak Memory Models (WMMs) such as ARMv8, Power, or RISC-V is even more challenging. WMMs allow hardware and compilers to aggressively reorder memory accesses, which greatly optimizes the performance of applications, but easily breaks the correctness of concurrent code.

In concurrent code, threads typically communicate via shared variables protected by synchronization primitives (*e.g.*, spinlocks, mutexes, read-write locks). Such concurrent code should work on WMMs out of the box [6] provided it contains no data races and the synchronization primitives are correct. Unfortunately, synchronization primitives are not always correct and break in subtle and non-reproducible ways when crucial memory operations are executed out of order.

To enforce some ordering among memory operations, WMMs provide *barriers*, which are either stand-alone explicit fences (*e.g.*, `DMB ISH` on ARMv8) or implicit barriers attached to memory operations (*e.g.*, `LDAR` and `STLR` on ARMv8). Such barriers need to be placed carefully inside the code so that the orderings required by the algorithm are enforced. However, as concurrent code often lies on the critical path, unnecessary or overly constrained barriers degrade the performance of the whole system. For this reason, experts spend a great deal of time and effort in identifying the key orderings among memory operations, and optimizing the usage of barriers accordingly [17, 18, 10, 16, 24].

Identifying the necessary order of memory operations is an error-prone task, even for experts. For example, the optimization of the barriers in the Linux `qspinlock` introduced a bug [16] that

remained unfixed for three years [10]. Despite the growing popularity of WMM architectures, new synchronization primitives are often published while omitting the required memory barriers. For example, Dice and Kogan note the importance of barriers and fences for the implementation of the Compact NUMA-Aware (CNA) lock:

> "Our actual implementation uses volatile keywords and memory fences as well as padding (to avoid false sharing) where necessarily." [11]

However, they do not provide detailed indications of where these barriers/fences should be placed. To work correctly on architectures such as ARMv8, Power, and RISC-V, practitioners must find and assign a safe and efficient sequence of barriers to the CNA implementation themselves.

Complementing the work of Dice and Kogan, we report in this technical note our efforts in verifying the correctness of CNA on WMMs and identifying where barriers/fences are necessary. The identification of barriers is based on the VSync framework [19], which we briefly review in Section 2. The verified and optimized CNA lock presented in Section 3 has been used in the evaluation of CLoF [8].

Since CNA is currently proposed as an alternative implementation of the slowpath of Linux `qspinlock` [9], we also investigate the correctness of Linux `qspinlock` and the latest Linux CNA patch (v15) on the LKMM, ARMv8, and Power memory models in Section 4. We found that Linux `qspinlock` and, consequently, Linux CNA are incorrect according to LKMM, but both are still correct under the ARMv8 and Power memory models.

2 VSync Overview

The VSync framework helps developers to implement, verify, and optimize concurrent code (*e.g.*, CNA lock) on WMMs. Here, we briefly highlight aspects relevant for this document — for details on VSync, please refer to our paper [19] and technical report [20].

Atomics. To facilitate the implementation of concurrent code, VSync provides a library of atomic operations and fences, but it also supports C11 atomics (via `stdatomic.h`). The library allows one to implement the atomic primitives with architecture-specific optimizations. Atomic libraries typically offer two types of barriers: *implicit barriers* (*i.e.*, barriers attached to atomic memory operations) and *fences* (*i.e.*, stand-alone barriers, also called *explicit barriers*). Currently, VSync accepts four barrier modes: **rlx** (*relaxed*), **acq** (*acquire*), **rel** (*release*), and **sc** (*sequentially consistent*). The semantics of barriers can roughly be understood as follows:

- no instruction can be reordered **before** an **acquire** barrier;
- no instruction can be reordered **after** a **release** barrier;
- a sequentially-consistent read implies an acquire barrier;
- a sequentially-consistent write implies a release barrier;
- a sequentially-consistent fences implies both an acquire barrier and a release barrier;
- sequentially-consistent accesses cannot be reordered with each other.

The relaxed mode **rlx** allows the processor to perform all memory-access optimizations, whereas the sequentially consistent mode **sc** disables most optimizations. The modes **rel** and **acq** are between the other two, allowing some optimizations. They are used to correctly implement efficient message-passing, producer/consumer patterns, and, of course, lock and unlock internals.

Verification. There are two key properties that synchronization primitives such as CNA lock should satisfy: *mutual exclusion*, *i.e.*, no two threads can execute the critical section (protecting the shared variables) at the same time; and *await termination*, *i.e.*, threads eventually leave the `lock_acquire()` and `lock_release()` functions. To verify these properties on WMMs, VSync employs a model checker — more specifically, AMC [19] integrated in GenMC [2, 15] v0.8 or later.

```c
#define N 5 // number of threads
cnalock_t lock;
cna_node_t node[N];
int v = 0; // shared state
void run(int id) {
    cna_lock(&lock, &node[id]);
    v++; // critical section
    cna_unlock(&lock, &node[id]);
}
void main() {
    pthread_t t[N];
    for (int i=0; i < N; i++)
        pthread_create(&t[i], 0, run, i);
    for (int i=0; i < N; i++)
        pthread_join(t[i], 0);
    assert (v == N); // check mutual exclusion
}
```

Figure 1: CNA client code used for checking and barrier optimization.

```
$ vsyncer compile cna-client.c -o cna-client.ll
$ vsyncer optimize -b -1 cna-client.ll
== SUMMARY ===================================
Barriers: 40
 Seq Cst: 2
 Acquire: 3
 Release: 4
 Relaxed: 31
== DIFF ======================================
[2] cnalock.h:38:
    atomicptr_write(&me->next, 0);
    ^~~~~~~~~ atomicptr_write_rlx
...
[7] cnalock.h:50:
    atomicptr_write(&tail->next, me);
    ^~~~~~~~~ atomicptr_write_rel
... (truncated)

```

Figure 2: VSync optimization report with maximally-relaxed barrier combination.

This model checker is aware of the VSync library and can therefore correctly reason about library calls.

VSync compiles to LLVM-IR the synchronization primitive together with a special client code that creates threads and drives the execution. The client code used with CNA lock (see Figure 1) increments a variable `v` in the critical section. Mutual exclusion is guaranteed if the model checker can show that `v` equals N at the end of all possible executions of the client code. The implementation is *live* (guarantees await termination) if the model checker can show that all possible executions of the client code terminate.

Optimization. *Barrier optimization* is the task of finding a combination of maximally-relaxed barrier modes for the concurrent code at hand, that still guarantees the two above key properties. This maximally-relaxed barrier combination allows the hardware to perform all memory-access optimizations that do not introduce property-violating behavior. Note that the combination cannot, however, enforce that the hardware indeed does optimizations and that the system performs better or is faster.

VSync optimizes barriers by gradually relaxing their modes, while employing the model checker at each step to verify the correctness of the current barrier combination. The order in which barriers are relaxed is decided by the adaptive linear relaxation algorithm, which allows VSync to quickly find a maximally-relaxed barrier combination and produce a report as in Figure 2. The reported barrier combination is guaranteed to be both correct and *maximally relaxed*, *i.e.*, further relaxing any of the barriers would cause the implementation to fail on WMMs.

Memory model. The default memory model is the intermediate memory model [22] (IMM), which unifies most dependency-tracking WMMs, including x86, Power, ARMv8, and RISC-V. We are currently working to support the Linux Kernel Memory Model [1] (LKMM) in VSync.

Caveat. Model checking can only analyze the correctness relative to the given client code. Our client code (Figure 1) is parametric in the number N of threads. It is up to the user to find a sufficiently large parameter N for which the client code can fully exercise the code under consideration and, in turn, produce a dependable verification and optimization result. We verified the CNA lock using 3, 4, and 5 threads. VSync finds the same maximally-relaxed barrier combination with 4 and 5 threads.

3 Verification and Optimization of CNA lock on WMMs

In this section, we report the results of our verification and optimization of the original CNA algorithm [11] on WMMs. We also discuss one example of how relaxing the resulting barrier

```
1  void cna_lock(cna_lock_t *lock, cna_node_t *me) {
2    me->next = 0;
3    me->socket = -1;
4    me->spin = 0;
5
6    cna_node_t * tail = SWAP_sc(&lock->tail, me);
7    if (! tail ) { me->spin = 1; return; }
8
9    me->socket = current_numa_node();
10   tail->next =_rel me;
11
12   while (!me->spin_acq) { CPU_PAUSE(); }
13 }
14
15 void cna_unlock(cna_lock_t *lock, cna_node_t *me) {
16   if (!me->next_acq) {
17     if (me->spin == 1) {
18       if (CAS_sc(&lock->tail, me, NULL) == me) return;
19     } else {
20       cna_node_t *secHead = (cna_node_t *) me->spin;
21       if (CAS_sc(&lock->tail, me, secHead->secTail) == me) {
22         secHead->spin =_rel 1;
23         return;
24       }
25     }
26     while (me->next == NULL) { CPU_PAUSE(); }
27   }
28   cna_node_t *succ = NULL;
29   if (keep_lock_local() && (succ = find_successor(me))) {
30     succ->spin =_rel me->spin;
31   } else if (me->spin > 1) {
32     succ = (cna_node_t *) me->spin;
33     succ->secTail->next = me->next;
34     succ->spin =_rel 1;
35   } else {
36     me->next->spin =_rel 1;
37   }
38 }
```

```
45 cna_node_t * find_successor(cna_node_t *me) {
46   cna_node_t *next = me->next;
47   int mySocket = me->socket;
48
49   if (mySocket == -1) mySocket = current_numa_node();
50   if (next->socket == mySocket) return next;
51
52   cna_node_t *secHead = next;
53   cna_node_t *secTail = next;
54   cna_node_t *cur = next->next_acq;
55
56   while (cur) {
57     if (cur->socket == mySocket) {
58       if (me->spin > 1)
59         ((cna_node_t *)(me->spin))->secTail->next = secHead;
60       else
61         me->spin = (uintptr_t )secHead;
62       secTail->next = NULL;
63       ((cna_node_t *)(me->spin))->secTail = secTail;
64       return cur;
65     }
66     secTail = cur;
67     cur = cur->next_acq;
68   }
69   return NULL;
70 }
```

Figure 3: Code implementing CNA Lock as presented in the original work [11], enriched with barrier annotations. The annotations **acq**, **rel** and **sc** respectively mean an acquire, release and sequentially-consistent implicit barrier. If the annotation is (1) after a = sign, it concerns a write access; (2) after an expression, it concerns a read access; or (3) after a function name, it concerns an atomic primitive. The absence of a barrier annotation means, for a struct field, a relaxed access, and for a non-shared stack variable, a non-atomic access.

combination can cause a bug, showing that the resulting combination is indeed *maximally-relaxed*.

Using the code provided in the original CNA work [11], we identified all concurrent accesses and replaced them by the corresponding atomic operations with **sc** barrier modes. That concerns not only the `SWAP` and `CAS` operations, but also every read and write access to the fields of `cna_node_t` and `cna_lock_t`. Figure 3 depicts the implementation with the resulting barrier relaxations, verified and optimized by VSync. For the sake of readability, we use a shortened notation to identify the necessary barrier modes: operations marked with subscripts **acq**, **rel**, and **sc** have the respective barrier mode. Other operations have no barrier attached (*i.e.*, **rlx**). These barriers guarantee mutual exclusion and await termination of the CNA locking algorithm on WMMs.

As mentioned above, we ran our client code configured with 3, 4, and 5 threads. We report the different checking and optimization times in Table 1.

Example of a Bug Caused by Overly-Relaxed Barriers. VSync guarantees the provided solution is *"maximally-relaxed"*, meaning any further relaxation could lead to a property-violating execution. To illustrate this point, we try to relax a barrier from the code depicted in Figure 3, and explain how it leads to data corruption. On Line 30, we remove the release barrier, fully relaxing the write to `succ->spin`. With that simple change, our VSync checker detects an *unsafe* execution with as few as two threads: mutual exclusion of the critical section is violated, allowing both threads to modify data concurrently.

To see the issue, let us use the following scenario where two threads T_0 and T_1, with a queue initially empty (`lock->tail == NULL`), run their code as depicted in Figure 1: both threads call

Table 1: Checker and optimizer time for CNA.

Threads	Verification time	Optimization time
3	1s	21s
4	1m31s	1m58s
5	1h49m54s	2h00m11s

`cna_lock`, increment a shared counter `v` (which is not of an atomic type), and call `cna_unlock`.

The critical section consists of an increment of the variable `v`. In the C language, this requires a *read* of the current value of `v` from memory, followed by a write of value `v+1` back to memory. If mutual exclusion of the critical section is violated, both threads could potentially read the same value `v` (in this case `v=0`), and both would then write `1` to the variable `v`.

Assume T_0 executes `cna_lock` first: it initializes `node[0]` (Lines 2-4), adds itself as the tail to the list using the $\texttt{SWAP}_{\texttt{sc}}$ operation (Line 6), and, as the list was empty, sets its own `node[0]->spin` to 1 and returns (Line 7), entering the critical section.

In parallel, T_1 executes `cna_lock` as well and adds itself as the tail after T_0; as T_0 is already in the queue, `node[0].next` is now `node[1]`, and T_1 starts to spin on its own `node[1]->spin` field (Line 12), waiting for T_0 to release it.

However, on T_0's side, the read of `v` in the critical section occurs (`v` evaluates to 0) but the write operation is delayed because there is no release barrier to avoid reordering it with later instructions. Indeed, the first test in `cna_unlock` fails (Line 16) as `node[0]->next` is not null but is equal to `&node[1]`, therefore the body of the `if` is skipped, and no barrier contained in it has any impact on that execution. After that, the execution jumps to Line 29: the function `keep_lock_local()` returns true (as it does most of the time), so `succ` is initialized to the return value of `find_successor(&node[0])`, which is `&node[1]`. No barrier in the `find_successor` function prevents further delaying the write of the critical section. Therefore, we enter the body of the `if`. Previously, Line 30 included a release barrier, thus forbidding to further delay the critical section write (of `v+1`). However, we relaxed it in this example, allowing the atomic write to `succ->spin` (Line 30) to occur first, thus releasing T_1 from spinning (Line 12). T_1 will then read `v` before T_0 had the chance to write `1` to it (thus violating mutual exclusion), so they both read `0`. Consequently, both threads will write `1` to `v`, resulting in data corruption for this specific execution. Our model checker (GenMC with AMC) is able to detect this execution as it violates an assertion that ensures the value of `v` to be correct (see client code in Fig. 1).

With this example, we showed that in an implementation with wrong barriers, subtle reorderings can occur and, combined with concurrent execution, can lead to data corruption. The safety property is thus violated.

We ran both this example and the correct version on an ARMv8 computer: in some cases, we observed data corruption in the former whereas we never observed any hang nor failure with the version that has correct and maximally-relaxed barriers (the one depicted in Figure 3).

4 Verification of Linux CNA on LKMM

The CNA lock is about to be merged into the Linux kernel as an alternative implementation for the slowpath of `qspinlock` [9]. The verification of the CNA lock in user space (Section 3) only partially supports the correctness of the CNA version being introduced in Linux. There are two reasons for that. First, the CNA algorithm has evolved to match Linux requirements. In particular, it is combined with a `qspinlock`, which in turn has its own memory barriers that can affect CNA. Second, the memory model used in VSync is IMM, but Linux has its own memory model, LKMM. As mentioned in Section 2, VSync does not yet fully support LKMM, so we decided to use the Dartagnan [3, 12] tool to verify (but not optimize) CNA under LKMM. Dartagnan is a Bounded Model Checking (BMC) tool that can find concurrency bugs in programs executed under various memory models, and in particular also under LKMM. However, it comes with minor limitations

which we discuss below.[1]

Since the CNA lock is combined with `qspinlock`, we first verified the correctness of the latter in isolation. During this process, Dartagnan found several correctness violations. According to LKMM, `qspinlock` guarantees neither mutual exclusion nor await termination, and it contains a data race. Below we discuss all these issues.

This might initially be shocking: *how is it possible that nobody noticed the main locking algorithm of the kernel to be fundamentally broken?* The reason for this is that the lock is incorrect according to LKMM, but all these issues disappear when the code is compiled and run on hardware. This discrepancy is due to the fact that language-level memory models (like LKMM) allow more flexible reorderings than hardware-level memory models (like ARMv8 and Power), which are effectively implemented in hardware.

Besides supporting code verification under LKMM, Dartagnan also allows the user to compile[2] the code to different architectures, and verify the resulting assembly output under the corresponding hardware memory model. Using Dartagnan we were able to prove the correctness[3] (both in terms of safety and await termination) of `qspinlock` under ARMv8 and Power.

The CNA lock inherits the correctness issues we found in `qspinlock`, making it incorrect as well under LKMM. However, using two changes described below, we were able to verify both `qspinlock` and CNA under LKMM. We next tried to verify the full CNA implementation under ARMv8 and Power. The verification of Linux CNA (patch v15 [14]) with 4 threads took less than 16 hours under LKMM and less than 12 hours under each targeted architecture (see Table 2 for detailed times for each verification case). We detailed on Github the procedure to reproduce the different runs of the model checker, together with a Linux kernel patch required to make it work [5]. As we describe next, we could only perform a partial verification of the Linux CNA due to some limitations in Dartagnan.

Dartagnan limitations. We faced a few challenges while verifying Linux CNA. First, Dartagnan does not support mixed-sized accesses. Linux `qspinlock` combines the fast spinlock with the tail of the MCS lock or CNA lock in a single 32-bit variable. This variable is accessed with atomic operations of multiple widths: times 32-bit, times 8-bits. Dartagnan cannot work with that. Second, the execution time of Dartagnan is exponential in the number of threads and the size of the code. Since CNA lock is used in the slow path only, more threads are required to exercise its code than when CNA is used as originally proposed (Section 3). The higher number of threads and the additional complexity introduced by other CNA changes drastically increase verification time.

CNA and `qspinlock` changes. To mitigate the limitations mentioned above, we removed the pending bit strategy of `qspinlock` and split the fast path spinlock and the CNA lock tail in two distinct 32-bit fields. We also changed the `qspinlock` algorithm as follows (see Figure 4a). A thread first acquires the slow path lock, *i.e.*, the CNA lock, as if the fast path spinlock acquisition would have failed. Once the thread has acquired the CNA lock, it reorders or flushes the CNA queue, as in the original Linux CNA patch. After that, the thread acquires the fast path spinlock (without pending bit logic). The remainder follows the original Linux CNA patch. The thread releases the CNA lock and enters the critical section while holding the fast path spinlock. After the critical section, the thread simply releases the fast path spinlock. In the CNA algorithm itself, the required changes were minimal (see Figure 4b). To simplify verification, we disable the shuffle reduction by returning always 0 from the `probably()` function. Moreover, we call `cna_order_queue()` only twice. Finally, the decision of when the intra-node threshold is reached is performed non-deterministically in the client code (Figure 4c). We expect that these changes do not affect the lock correctness whereas they reduce the complexity of the implementation such that verification times become reasonable.

[1] Note that a previous version of this technical note [21] reported in this section results using the GenMC tool. Unfortunately, those results were incorrect due to code simplifications and model checker issues with LKMM. The results presented here supersede the previous results.

[2] Dartagnan follows the kernel implementation of atomics, *i.e.*, located in the kernel source tree in `arch/<target-arch>/include/asm/atomic.h`.

[3] As Dartagnan is a BMC tool, loops are unrolled. The correctness holds up to the unrolling bound.

```c
typedef struct qspinlock {
  int spinlock;     /* cmpxchg-based spin lock */
  union {
    atomic_t val;
        /* val is only CNA tail
         * not a union with locked and pending */
    };
};

void
queued_spin_lock_slowpath(struct qspinlock *lock, u32 val)
{
  struct mcs_spinlock *prev, *next, *node;
  u32 old, tail;
  int idx;
  /* completely removed lock-pending logic */

queue:
  node = this_cpu_ptr(&qnodes[0].mcs);
  idx = node->count++;
  tail = encode_tail(smp_processor_id(), idx);
  node = grab_mcs_node(node, idx);

  barrier();

  node->locked = 0;
  node->next = NULL;
  cna_init_node(node);

  smp_wmb();

  old = xchg_tail(lock, tail);
  next = NULL;

  if (old & _Q_TAIL_MASK) {
    prev = decode_tail(old);
    WRITE_ONCE(prev->next, node);

    arch_mcs_spin_wait(&node->locked);

    next = READ_ONCE(node->next);
    if (next)
      prefetchw(next);
  }
  if ((val = cna_wait_head_or_lock(lock, node)))
    goto locked;

  /* removed lock pending logic in the next read */
  val = atomic_read_acquire(&lock->val);

locked:
  /* acquire spinlock after acquiring CNA lock */
  await_while(cmpxchg_acquire(&lock->spinlock, 0, 1) != 0);

  if (cna_try_clear_tail(lock, val, node))
    goto release;

  if (!next)
    next = smp_cond_load_relaxed(&node->next, (VAL));

  cna_lock_handoff(node, next);

release:
  __this_cpu_dec(qnodes[0].mcs.count);
}
bool __try_clear_tail(struct qspinlock *lock,
              u32 val,
              struct mcs_spinlock *node)
{
  /* removed pending bits logic */
  return atomic_try_cmpxchg_relaxed(&lock->val, &val, 0);
}
void queued_spin_unlock(struct qspinlock *lock)
{
  /* release spinlock */
  smp_store_release(&lock->spinlock, 0);
}

```

(a) Modified fast path in `qspinlock.{c,h}`

```c
bool probably(unsigned int num_bits)
{
  return 0;
}

bool intra_node_threshold_reached(struct cna_node *cn)
{
  return my_threshold != 0;
}

u32 cna_wait_head_or_lock(struct qspinlock *lock,
            struct mcs_spinlock *node)
{
  // ... (truncated)
  if (!cn->start_time || !intra_node_threshold_reached(cn))
    {
    if (cn->numa_node == CNA_PRIORITY_NODE)
      cn->numa_node = cn->real_numa_node;

    /* forced only two cna_order_queue call by replacing
     loop:
     *  while (LOCK_IS_BUSY(lock) && !cna_order_queue(node))
     *    cpu_relax();
     * with these calls: */
    cna_order_queue(node);
    cna_order_queue(node);

  } else {
    cn->start_time = FLUSH_SECONDARY_QUEUE;
  }
  return 0;
}

```

(b) Minor changes in `qspinlock_cna.h`

```c
#define N 4 // number of threads
struct qspinlock lock;
struct cna_node my_node[N];
int my_threshold = 0;
int x = 0; // first counter
int y = 0; // second counter
void run(int id) {
  queued_spin_lock_slowpath(&lock, 0);
  // critical section
  x = x + 1;
  y = y + 1;
  queued_spin_unlock(&lock);
}
void main() {
  pthread_t t[N];
  for (int i=0; i < N; i++)
    pthread_create(&t[i], 0, run, i);
  my_threshold = 1; // non-deterministically decide when
                    // threshold is reached
  for (int i=0; i < N; i++)
    pthread_join(t[i], 0);
  assert (x == y);  // check of mutual exclusion
}
```

(c) Client code for Linux CNA

Figure 4: Linux CNA and `qspinlock` modifications and client code to verify Linux CNA with Dartagnan on LKMM, ARMv8 and Power.

```
// CS + releasing lock
P0(intptr_t *x, intptr_t *lock)
{
        WRITE_ONCE(*x, 1);
        atomic_set_release(lock, 1);
}

// xchg_tail
P1(intptr_t *x, intptr_t *l)
{
    int val = atomic_read(lock);
    int new = val + 2;
    int old = atomic_cmpxchg_relaxed(lock, val, new);
}

// acquiring lock + CS
P2(intptr_t *x, intptr_t *lock)
{
    int r0 = atomic_read_acquire(lock);
    int r1 = READ_ONCE(*x);
}

exists (2:r0=3 /\ 2:r1=0)
```

```
// node initialization in the slowpath
P0(intptr_t *nodeLocked, intptr_t *tailNext)
{
    *nodeLocked = 1;
    smp_wmb();
    WRITE_ONCE(*tailNext, 1);
}

// mcs_lock_handoff
P1(intptr_t *nodeLocked)
{
    smp_store_release(nodeLocked, 2);
}
```

Figure 5: Correctness violations in `qspinlock`. Safety (left) and a data race (right).

Correctness violations in `qspinlock`. DARTAGNAN found three issues with `qspinlock` under LKMM: it does not guarantee mutual exclusion (safety violation), it contains a data race (also confirmed by GENMC), and it can hang (await termination violation). To showcase these issues, we present litmus tests that isolate the problematic behaviors and hence can be analyzed, and thus confirmed, by the HERD7 tool [4]. The safety violation (left of Figure 5) is due to a lack of synchronization in a release-acquire chain. The intended synchronization between P0 releasing the lock and P2 acquiring the lock is broken if thread P1 interferes using a relaxed RMW operation like the CAS in `xchg_tail`. This is because the LKMM, unlike the C11 memory model, does not maintain release-acquire chains over relaxed RMW operations [23]. Changing the relaxed CAS to a release one (referred in Table 2 as *"fix 2"*) solves this issue. The await termination violation is due to a missing propagation which can make `arch_mcs_spin_wait` loop forever. While the `wmb()` in Line 30 is enough to guarantee that the nodes are correctly initialized before changing the tail, in the LKMM, `wmb()` does not guarantee necessary propagation properties between different threads. This problem can be solved by using `mb()` instead of `wmb()`. The data race (right of Figure 5) occurs between the plain write to initialize the MCS node (Line 26 in Figure 4a) and the write in `cna_lock_handoff` (Line 61) which is implemented using a `smp_store_release`. There are two possible solutions to remove this race, either initialize the node using `WRITE_ONCE`, or use `mb()` instead of `wmb()` after the node initialization. We opted for the latter since this single change (referred in Table 2 as *"fix 1"*) solves both the await termination and data race problem.

With these two changes, DARTAGNAN finds no more issues in the core of `qspinlock`. However, it still reports data races in the critical section of the client code. We deem those races to be spurious, because the fixed `qspinlock` guarantees mutual exclusion and, hence, there cannot be a data race in the critical section according to any intuitive definition of data race. As a result, we believe the data race definition of LKMM is broken.

Experimental Setup. The evaluation was executed on a GIGABYTE **R182-Z91-00** server [13] equipped with 2 **AMD EPYC 7352** 24-Core Processor [7] and 125 GiB of RAM. Table 2 reports the verification time, the verification result and what kind of violations are found (if applicable).

5 Conclusion

We have reported our efforts in verifying and optimizing the CNA lock on WMMs. Using VSYNC, we derived a safe and efficient set of barriers (with their associated memory order modes). Using a client code with 3, 4, and 5 threads, we showed that the implementation is correct on WMMs with the maximally-relaxed sequence of barriers.

Table 2: DARTAGNAN verification times for `qspinlock` and CNA, with and without the applied fixes. We also report the verification results and what kind of violations are found.

Memory model	Lock algorithm	Verification time	Verified?	Violation type
LKMM	`qspinlock`, unmodified	37 s	✗	Liveness
LKMM	`qspinlock`, with fix 1	2 min	✗	Safety
LKMM	`qspinlock`, with fix 1 and 2	2 min	✓	
ARMv8	`qspinlock`, unmodified	3 min	✓	
POWER	`qspinlock`, unmodified	3 min	✓	
LKMM	CNA, unmodified	27 min	✗	Liveness
LKMM	CNA, with fix 1 and 2	15h 26 min	✓	
ARMv8	CNA, unmodified	11h 42 min	✓	
POWER	CNA, unmodified	11h 23 min	✓	

We also studied the CNA patch that is currently under review to be integrated into the Linux kernel. Although we could not run our optimization on the patch, we could analyze it by applying few simplifications to the `qspinlock` code. We proved that the patch (v15) is correct under ARMv8 and POWER. We described the violations found by DARTAGNAN under LKMM and explained how to fix them.

References

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

[2] GENMC, 2021. `https://github.com/MPI-SWS/genmc`.

[3] DARTAGNAN, 2022. `https://github.com/hernanponcedeleon/Dat3M`.

[4] HERD7, 2022. `https://github.com/herd/herdtools7`.

[5] Verification of Linux qspinlock_cna, 2022. `https://github.com/huawei-drc/cna-verification/tree/arxiv-cna-wmm-v2-20220709`.

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

[7] AMD. 2nd Gen AMD EPYC™ 7352 | Server Processor | AMD. `https://www.amd.com/en/products/cpu/amd-epyc-7352`. Accessed: 2022-07-08.

[8] Rafael Chehab, Antonio Paolillo, Diogo Behrens, Ming Fu, Hermann Härtig, and Haibo Chen. CLoF: A Compositional Lock Framework for Multi-Level NUMA Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 851–865, New York, NY, USA, 2021. Association for Computing Machinery.

[9] Jonathan Corbet. NUMA-aware qspinlocks [LWN.net], 2021. `https://lwn.net/Articles/852138/`.

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

[11] Dave Dice and Alex Kogan. Compact NUMA-Aware Locks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[12] Natalia Gavrilenko, Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. BMC for weak memory models: Relation analysis for compact SMT encodings. In *CAV*, volume 11561 of *LNCS*, pages 355–365. Springer, 2019.

[13] GIGABYTE. R182-Z91 (rev. 100) | Rack Servers - GIGABYTE Global. `https://www.gigabyte.com/Rack-Server/R182-Z91-rev-100`. Accessed: 2022-07-08.

[14] Alex Kogan. [PATCH v15 0/6] Add NUMA-awareness to qspinlock, 2021. `https://lore.kernel.org/linux-arm-kernel/0FF6E325-3490-4A8B-BB04-D36CCBFA1D19@oracle.com/T/`.

[15] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Model checking for weakly consistent libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 96–110, New York, NY, USA, 2019. Association for Computing Machinery.

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

[17] Waiman Long and Peter Zijlstra. qspinlock code at version 4.4 of Linux Kernel, 2015. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/kernel/locking/qspinlock.c?h=v4.4`.

[18] Waiman Long and Peter Zijlstra. qspinlock code at version 5.6 of Linux Kernel, 2020. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/kernel/locking/qspinlock.c?h=v5.6`.

[19] 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. 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, New York, NY, USA, 2021. Association for Computing Machinery.

[20] 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. VSync: Push-Button Verification and Optimization for Synchronization Primitives on Weak Memory Models (Technical Report), 2021.

[21] Antonio Paolillo, Diogo Behrens, Rafael Lourenco de Lima Chehab, and Ming Fu. Verifying and optimizing compact numa-aware locks on weak memory models (v1). *CoRR*, abs/2111.15240v1, 2021. `https://arxiv.org/abs/2111.15240v1`.

[22] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. Bridging the gap between programming languages and hardware weak memory models. *Proceedings of the ACM on Programming Languages*, 3(POPL), January 2019.

[23] Alan Stern. `https://github.com/paulmckrcu/litmus/issues/11`, 2022. Accessed: 2022-06-27.

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