Verifying and Optimizing the HMCS Lock for Arm Servers

Jonas Oberhauser[1,2], Lilith Oberhauser[1,2], Antonio Paolillo[1,2], Diogo Behrens[1,2], Ming Fu[1,2], and Viktor Vafeiadis[3]

[1] Huawei Dresden Research Center, 01067 Dresden, Germany
[2] Huawei OS Kernel Lab, China
firstname.lastname@huawei.com
[3] Max Planck Institute for Software Systems, 67663 Kaiserslautern, Germany
viktor@mpi-sws.org

Abstract. To optimize the performance of some of our systems running on non-uniform memory architecture (NUMA) servers with Arm processors, we have implemented multiple versions of the HMCS lock, an advanced NUMA-aware lock that has been identified in the literature as particularly scalable.
This is a highly non-trivial task because of the many implementation choices for interlocked operations, alignment, and memory barrier placement, affecting not only the lock's performance but also its correctness. The published HMCS lock does not discuss choices that affect performance, but it does present a choice of barriers. We observe that this choice is wrong, leading to hangs on Kunpeng Arm servers. We repair the barriers and implement the first formally-verified HMCS lock with VSync, an automated formal verification and optimization tool for weak consistency. We explain the barrier bugs in detail and report our experience of barrier optimizations for Arm servers.

Keywords: Consistency Models · Verification · Optimization · NUMA-aware locks

1 Introduction

Arm is making inroads on many-core servers [11,4]. To achieve a high level of parallelism, these many-core servers are implemented as non-uniform memory architectures (NUMA) in which CPUs are clustered on NUMA nodes. In these architectures, communication between CPUs within a single node is much faster than across nodes. Software therefore needs to ensure *locality* to scale well, i.e., avoid communication across NUMA nodes.

One strategy to achieve locality is through so-called NUMA-aware locks, which favor CPUs within the same NUMA node when passing the lock. Among these, we have chosen the NUMA-aware HMCS lock [6], which has been shown to be very scalable [5,9]. We have implemented the NUMA-aware HMCS lock on Arm with the goal of improving the performance of Huawei products running on

Kunpeng Arm servers. Implementing the HMCS lock for use in industry involved two main challenges.

The first challenge is the *weak consistency*. To improve single-core performance, Arm CPUs commit and propagate memory operations out-of-order: for example, memory operations issued after a cache miss can be performed while the missing cache line is being fetched. Such optimizations can be fatal to the HMCS lock, which relies on the order of a few crucial memory operations. To avoid bugs, one needs to selectively turn these optimizations off through so-called memory barriers; these include stand-alone explicit fences (e.g., `DMB`) as well as implicit barriers attached to the memory operations (e.g., `LDAR` and `STLR`). Turning off the optimizations everywhere is relatively easy, e.g., by using sequentially consistent C11 atomics to insert barriers for every memory access. The excessive use of barriers, however, does degrade performance. Therefore, experts attempt to identify precisely the operations that need to be executed in-order, and insert only barriers needed to enforce those orders. Indeed, the original HMCS lock paper "shows the fences necessary for the HMCS lock on systems with processors that use weak ordering" [6, p. 218], as identified by its authors. Our investigation reveals that these fences are wrong, potentially leading to hangs on Arm, Power, and RISC-V. We have reproduced the hang on a Kunpeng Arm server.

The second challenge is *performance-tuning*. We investigate two main factors that influence the performance of the HMCS lock: (1) the implementation of atomic `SWAP` and `CAS` operations and (2) the placement of barriers. These atomic operations can be implemented on Arm either through built-in interlocked `SWP` and `CAS` instructions (introduced in Arm's LSE extension [8]), which perform the operation in memory, or with load/store-exclusive `LDXR`/`STXR` instruction pairs, which perform the operation inside the CPU. For barrier placement there are similarly various implementation choices, e.g., between fences and implicit barriers. As the performance implications of these choices are not well-understood, the best choice needs to be identified by trial-and-measure.

In this paper, we show how to solve both challenges with the help of VSync [15], a formal verification and optimization tool for weak consistency. We generate formally verified barrier placements with VSync (Fig. 2). Since precise Arm support is not yet implemented in VSync, our barriers are verified against the slightly weaker IMM (intermediate memory model [16]) model, which forms the least common denominator of several weak consistency models including Arm, Power, and RISC-V. Thus the verified barriers are correct but not optimal for Arm. In fact, VSync detects a second hang and a mutual exclusion violation on IMM, but we manually verify that these bugs cannot occur on Arm weak consistency.

In the following, we present the HMCS algorithm (Section 2) and discuss the set of barriers necessary for its correctness (Section 3), showing what goes wrong if some barriers are omitted. We then briefly describe our verification and optimization setup (Section 4). Finally, we measure the performance impact of the implementation choices mentioned above as well as the conservative barriers introduced by VSync (Section 5) on a microbenchmark and on LevelDB [7].

In summary, we make the following contributions:

- We have discovered a bug in the fences proposed in the HMCS lock from the literature, and present a formally verified fix.
- We propose various barrier optimizations for the HMCS lock and investigate their impact on performance.
- We present the following insights:
  - Barriers optimizations make little difference for scalability; sequentially consistent C11 atomics are good enough for Arm.
  - If barrier optimizations are desired, they should be left to an automatic tool like VSync.
  - Arm's interlocked instructions (LSE) degrade performance.

2 Background

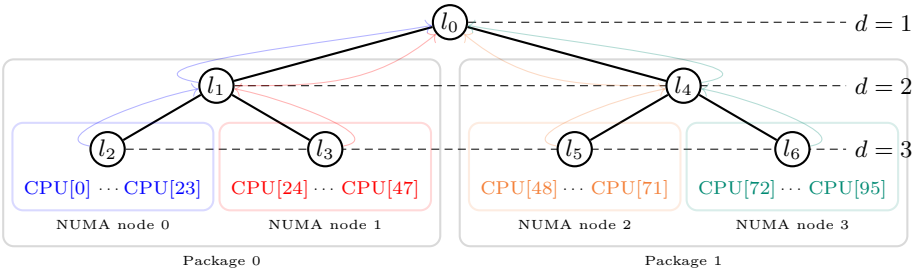


Fig. 1: NUMA topology and lock trees for 96-core Kunpeng Arm server

2.1 HMCS Lock

The HMCS lock is a tree of MCS locks, configured to model the NUMA topology tree of the target machine; in our case, we consider a Kunpeng 920 Arm server with four NUMA nodes (24 CPUs each), organized in two packages. As illustrated in Fig. 1, the lock tree for this topology is a binary tree of depth `DEPTH=3`. We now explain the MCS lock, which is the main component of the HMCS lock, and the acquire and release protocols of the HMCS lock. The code of the HMCS lock is shown in Fig. 2.

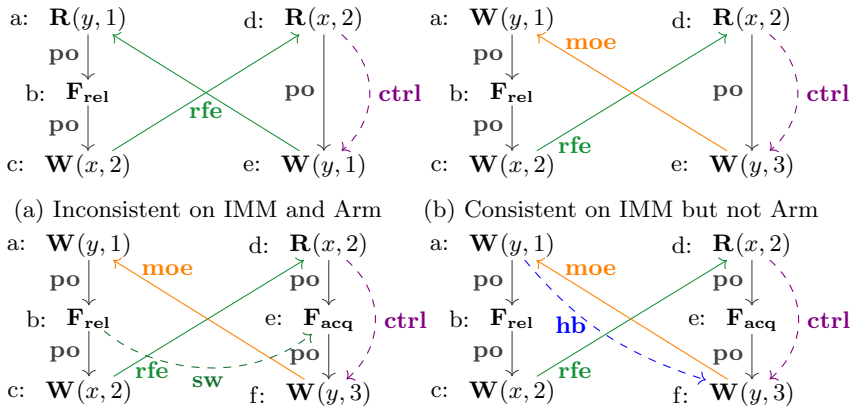
MCS Lock. The MCS lock [14] forms a queue so that threads enter the critical section in a FIFO manner. Acquiring and releasing the MCS lock are performed by the `AcqReal<1>` and `RelReal<1>` functions in Fig. 2. A thread enqueues its `QNode`, which contains a `status` field that is used as means of communication with its predecessor. Before enqueueing, a thread sets its `status` to 🔒 (Line 14),

```
1  enum LockStatus {
2    ↥=UINT64_MAX-1,
3    🔓=0x1,
4    🔒=0x0,
5    n ∈ [2 : THRESHOLD]
6  };
7
8  Acquire(HNode *L, QNode *I){
9    AcqReal<DEPTH>(L, I);
10   ---- ACQUIRE FENCE ----
11 }
12
13 AcqReal<1>(HNode *L, QNode *I){
14   I->status = 🔒; I->next = ⊥;
15   ---- RELEASE FENCE ARMIMM ----
16   QNode *pred;
17   pred = SWAP_sc(& L->tail, I);
18   if (!pred) {
19     I->status = 🔓;
20   } else {
21     pred->next =_rel I ;
22     while (I->status_acq == 🔒);
23   }
24 }
25
26 AcqReal<d>(HNode *L, QNode *I) {
27   I->status = 🔒; I->next = ⊥;
28   ---- RELEASE FENCE ----
29   QNode *pred;
30   pred = SWAP_sc(& L->tail, I);
31   if (pred) {
32     pred->next =_rel I;
33     LockStatus curStatus;
34     do curStatus = I->status_acq
35     while (curStatus == 🔒);
36     ---- ACQUIRE FENCE IMM ----
37     if (curStatus < ↥) return;
38   }
39   I->status = 1;
40   AcqReal<d-1>(L->parent, & L->N);
41 }
42
43 Release(HNode *L , QNode *I){
44   ---- RELEASE FENCE ----
45   RelReal<DEPTH>(L, I);
46 }
47
48 ReleaseHelper(HNode *L, QNode *I,
49   LockStatus st) {
50   QNode *succ = I->next_acq;
51   ---- ACQUIRE FENCE IMM ----
52   if (succ) {
53     succ->status =_rel st;
54   } else {
55     if (CAS_sc(& L->tail, I, ⊥))
56       return;
57     while ((succ = I->next) == ⊥);
58     succ->status =_rel st;
59   }
60 }
61
62 RelReal<1>(HNode *L, QNode *I){
63   ReleaseHelper(L, I, 🔓);
64 }
65
66 RelReal<d>(HNode *L , QNode *I){
67   uint64_t curCount = I->status;
68   if (curCount == THRESHOLD[d]) {
69     RelReal<d-1>(L->parent, & L->N);
70     ---- RELEASE FENCE ----
71     ReleaseHelper(L, I, ↥);
72     return;
73   }
74   QNode *succ = I->next_acq;
75   ---- ACQUIRE FENCE IMM ----
76   if (succ) {
77     curCount += 1;
78     succ->status =_rel curCount;
79     return;
80   }
81   RelReal<d-1>(L->parent, & L->N);
82   ---- RELEASE FENCE ----
83   ReleaseHelper(L, I, ↥);
84 }
```

Fig. 2: Pseudo-code of the HMCS Lock from [6] except for barrier placement and cosmetic changes

then it advances the tail pointer (Line 17). If it finds a predecessor p it waits in Line 22 for p to give the signal `status` $= 🔓$; otherwise it unlocks itself (Line 19) and enters the critical section. Once it is done, it releases the lock. If it is the tail (i.e., it has no successor), it does so by setting the tail pointer to $\perp$ (Line 55). Otherwise, if it has a successor s, it signals s by setting the `status` of s to 🔓 (Line 53 or Line 58).

HMCS Lock Acquisition. The critical section is protected by the root lock l_0 at depth $d = 1$. To initiate the lock acquisition protocol, the HMCS lock client calls `Acquire` on the leaf lock that belongs to the NUMA on which the thread is running; e.g., in Fig. 1 a thread bound to CPU[0] calls `Acquire` on l_2. This calls `AcqReal<3>` on l_2, which recursively calls `AcqReal<2>` on l_1 (l_2's parent)



(a) Inconsistent on IMM and Arm

(b) Consistent on IMM but not Arm

(c) Inconsistent on IMM and Arm

(d) (Fig. 3(c)) with **hb** shown and **sw** omitted

Fig. 3: Execution graphs

and `AcqReal<1>` on l_0 (l_1's parent). Note that this means a) all threads in the queue of a leaf lock are on the same NUMA node, b) all threads in the queue of a lock at depth $d = 2$ are in the same package but different NUMA nodes, and c) all threads in the queue of lock l_0 are in different packages. Enqueueing at any MCS lock in the tree requires a `QNode`. Each thread T_i has its own `QNode` N_i with which it enqueues at its leaf lock. Each lock l protects a `QNode` l.`N` which is used to enqueue at the parent of l. For example, if T_1 is running on NUMA node 0, it uses N_1 to enqueue at l_2. Once it owns that lock, it can use l_2.`N` to enqueue at l_1, and so on.

HMCS Lock Release. Invoking `Release` initiates the release protocol. This recursively calls `RelReal<3>`, `RelReal<2>` and `RelReal<1>`. The lock can be passed at any depth $d \in \{1, 2, 3\}$, if a successor is found at depth d. To maximize throughput, the lock should be passed within the NUMA node, i.e., at depth $d = 3$. However, this would lead to starvation in the other NUMA nodes. `THRESHOLD`$[d]$ defines the maximum number of times a lock is passed at depth $d \in \{2, 3\}$. If `THRESHOLD`$[d]$ has been reached, the lock owner sets the `status` of the successor at depth d to ↥. This signals to the successor that the lock is passed at a depth $d' < d$. In contrast, when a successor is found and `THRESHOLD`$[d]$ is not reached, the lock is passed directly to the successor by setting its `status` to $n \in [2 : \texttt{THRESHOLD}[d]]$, counting the number of times the lock was acquired at depth d.

2.2 Weak Consistency and Execution Graphs

A standard way to define weak consistency models is through execution graphs such as those in Fig. 3. **Nodes** in these graphs represent events such as reads

($\mathbf{R}$, $\mathbf{R_{sc}}$, $\mathbf{R_{acq}}$), fences ($\mathbf{F_{acq}}$ and $\mathbf{F_{rel}}$), and writes ($\mathbf{W}$, $\mathbf{W_{sc}}$, $\mathbf{W_{rel}}$), and **edges** specify various relations between these events, such as **moe** (**m**odification **o**rder **e**xternal) and **rfe** (**r**eads-**f**rom **e**xternal) edges which indicate the order in which reads and writes to the same location are committed, and **po** (**p**rogram **o**rder) edges which indicate the order in which instructions are issued (but not necessarily committed). In this paper, we only give high-level explanations for differences between Arm and IMM; motivated readers will find more detailed explanations in Appendix A. A weak consistency model is defined by the execution graphs it permits. For IMM and Arm, this is done by forbidding graphs in which any event "happens before" itself, where "happens before" is defined by model-specific relations that indicate the order in which events happen. The difference between the models can be explained in terms of when one event "happens before" another according to the model.

Arm has one "happens before" relation (called **ob**, **o**rdered-**b**efore), which respects among other things: a) the order in which writes are committed (**moe**), b) the order between a write and a read that observes the write (**rfe**), c) fences such as `DMB.ISH` (implied by a release fence ($\mathbf{F_{rel}}$) in the code), and d) control dependencies from a read influencing the position of control, e.g., through an if-condition, to a write occurring after the condition (written **ctrl** in graphs). In Figs. 3(a) and 3(b), this means that Event a "happens before" itself on Arm.

On IMM, there are two "happens before" relations, both weaker than that of Arm. A graph is forbidden if an event "happens before" itself according to either relation. The first is the **a**cyclic **r**elation (**ar**) which critically does not respect **moe**. According to this relation, Event a "happens before" itself only in Fig. 3(a), not in Fig. 3(b). The second relation (which is nameless in IMM, but which we will call $\mathbf{hb}_{\mathrm{IMM}}$) respects **moe**, but critically ignores control dependencies; thus according to this relation, Event a "happens before" itself neither in Figs. 3(a) and 3(b). Indeed, none of the other events "happen before" themselves in Fig. 3(b) with either definition, and Fig. 3(b) is consistent on IMM.

Unfortunately, behaviors like that in Fig. 3(b) lead to various bugs in the HMCS lock. To forbid this behavior on IMM, one has to add an acquire fence $\mathbf{F_{acq}}$ along the **ctrl** edge (Fig. 3(c)). The existing $\mathbf{F_{rel}}$ fence **s**ynchronizes-**w**ith (**sw**) this $\mathbf{F_{acq}}$, creating a **h**appens-**b**efore (**hb**) edge from Event a to Event f (Fig. 3(d)); together with the **moe** edge in the opposite direction, Event a "happens before" itself according to $\mathbf{hb}_{\mathrm{IMM}}$. Thus Fig. 3(c) is inconsistent on IMM.

3 Barriers on Arm and IMM

Figure 2 shows two formally verified barrier placements: one uses the highlighted implicit barriers, and the other uses the highlighted fences. Both use sequentially consistent (**sc**) `SWAP` and `CAS` operations. Further barrier optimizations on these operations are possible but bring no performance benefit (see hmcs-amo in Fig. 10 on Page 12, or a detailed discussion in Appendix B) and are thus not shown.

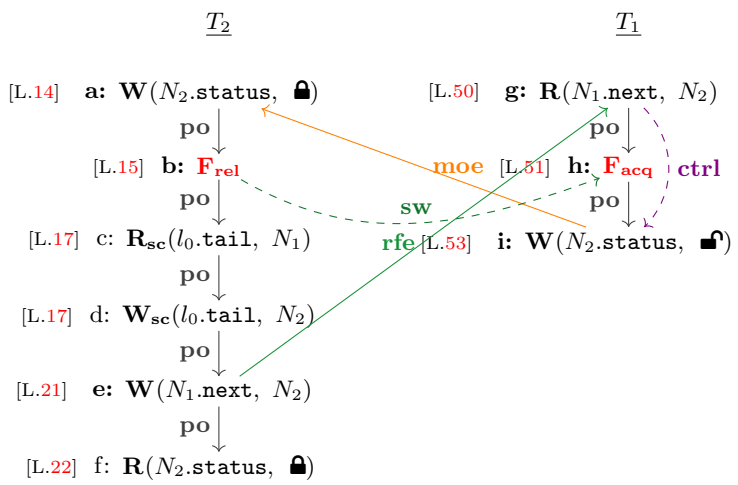


Fig. 4: Bug on Arm and IMM: Non-terminating execution due to missing fences at Events b and h

In addition to the fences already presented in [6], VSync introduces fences at Lines 36, 51 and 75 (for IMM) and Line 15 (for IMM and Arm) to solve three bugs. In the following sections we discuss these in more detail.

3.1 Termination Violation

To simplify the discussion of the hang we consider only an HMCS lock L with maximum depth one (`DEPTH` = 1), with two threads T_1 and T_2. We first discuss the desired behavior in which the lock is passed correctly. Initially thread T_1 owns the lock and is about to release it, while thread T_2 is attempting to acquire the lock. T_2 first prepares its node (Line 14), writing 🔒 to its `status` to indicate that it does not yet have permission to enter the critical section. It proceeds to append itself to the queue by moving the tail pointer (Line 17) and updating T_1's `next` pointer (Line 21). T_1 sees its successor in Line 50 or after failing to set the tail pointer to $\perp$ in Lines 55 and 57. Subsequently T_1 will set T_2's `status` to 🔓, indicating that T_2 can enter the critical section (Line 58).

On Arm, T_2's initialization to its own node can happen *after* it informs T_1 that it has a successor. In this case T_1 can unlock T_2 before T_2 initializes its node (locking itself again). An execution graph for this case is shown in Fig. 4. Perhaps surprisingly, the `SWAP` operation in Line 17 does not prevent this reordering even if it is sequentially consistent (note that the original presentation in [6] does not mention whether the atomic `SWAP` and `CAS` operations have any ordering semantics). The reason for this is that sequentially consistent atomic operations are compiled to `LDAXR`/`STLXR` instruction pairs which generate the Events c and d. Intuitively speaking, Arm only preserves the order 1) between Event c and subsequent events, 2) between Event d and preceding events, and

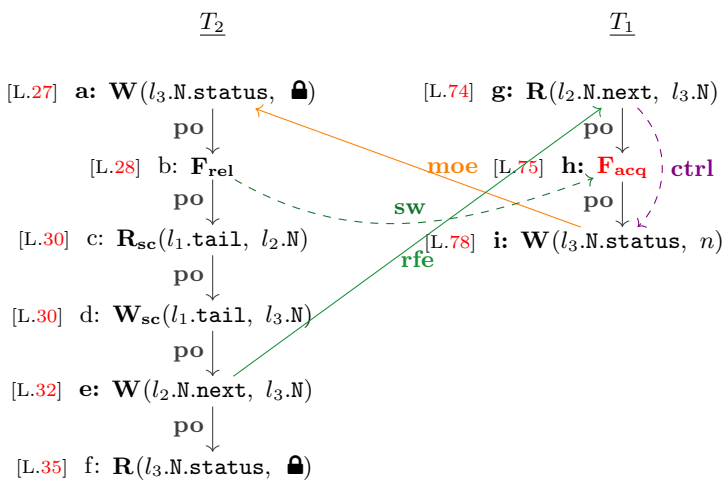


Fig. 5: Bug on IMM: Non-terminating execution

3) between all sequentially consistent events on the same processor. But, it does not preserve the order between Event a and Event c, or Event d and Event e. Thus the events can be committed in the order c, e, a, d. In this commit order, the node initialization (Event a) happens after T_2 informs T_1 (Event e).

We repair the bug by adding a $\mathbf{F_{rel}}$ fence in Line 15 in `AcqReal<1>`. With this fence, the Events a, b, e, g and i map directly to the events in Fig. 3(b). Thus (with the fence) the buggy execution becomes inconsistent on Arm, and the bug can not occur anymore. On IMM we additionally need to add an $\mathbf{F_{acq}}$ fence at Event h. With both fences, the events Events a, b, e and g to i map directly to the events in Fig. 3(c), showing that the bug is fixed also on IMM.

Note that for higher depths $d > 1$, the corresponding $\mathbf{F_{rel}}$ fence already exists (in Line 28), but the corresponding $\mathbf{F_{acq}}$ fence is also missing. Indeed VSync reports the analogous termination bug (Fig. 5) at greater depths. Analogously to before, we can see that this bug only exists on IMM and that it can be fixed by inserting the $\mathbf{F_{acq}}$ fence in Line 75.

3.2 Mutual Exclusion Violation

This bug only occurs with three threads T_1, T_2 and T_3 on separate NUMA nodes, as indicated in Fig. 6. In a nutshell, T_2 enqueues behind T_3 at l_0 with the `QNode` l_1.N, which was previously used by T_1 (Fig. 7(b)). When T_1 entered the critical section (Fig. 7(a)), it had no predecessor and therefore set the `status` of l_1.N to 🔓 (Line 19). Due to a missing fence, this operation is only propagated to T_2 after T_2 enqueued behind T_3, giving T_2 the false signal that it can enter the critical section even though T_3 is still holding the lock (Fig. 7(c)).

A more detailed execution leading to the bug is shown in Fig. 8. Note that Events d, e, g, i and l map to the events in Fig. 3(b), implying that the bug is not

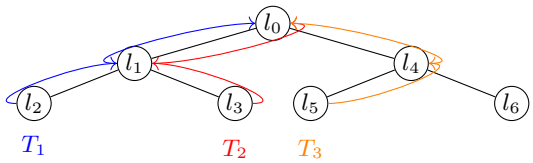
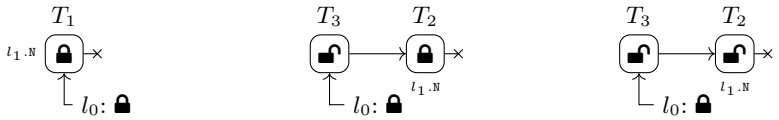


Fig. 6: Assignment of threads to NUMAs



(a) T_1 has no predecessor, but does not set `status` of l_1.N to 🔓 yet.

(b) T_3 enqueues. T_1 passes the lock to T_3. Later T_2 enqueues with l_1.N.

(c) T_1's update to l_1.N's `status` is finally propagated to T_2, unlocking it.

Fig. 7: Mutual exclusion violation on IMM due to a missing acquire fence.

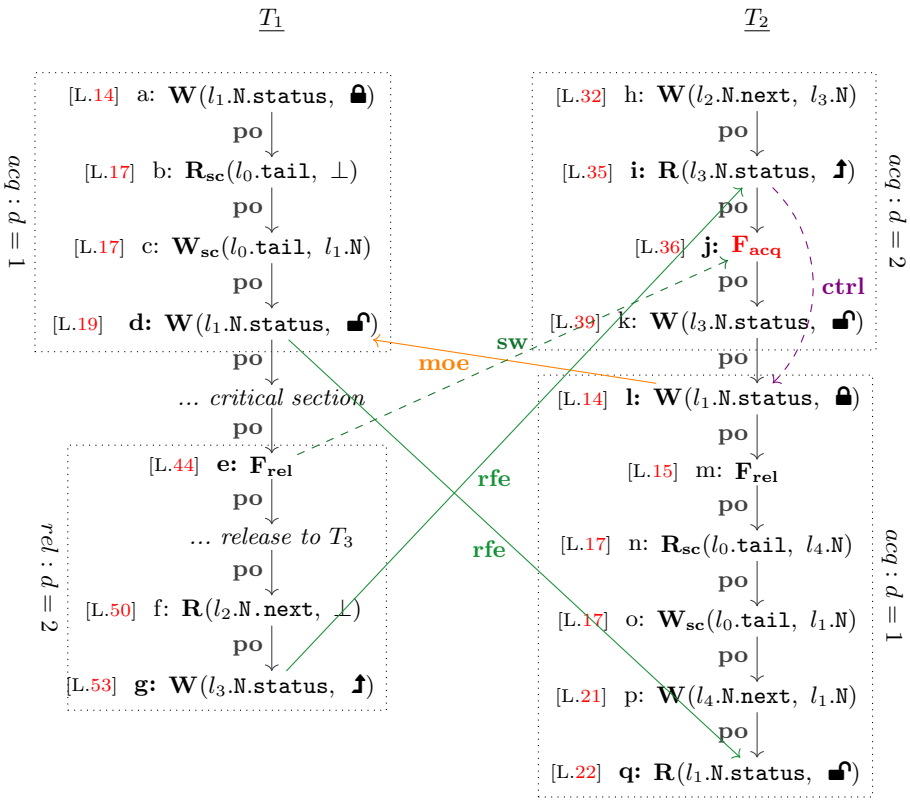


Fig. 8: Bug on IMM: Mutual exclusion violation execution graph.

```
        T1                          T2                          T3
Acquire(l2, N1);            Acquire(l3, N2);            Acquire(l5, N3);
counter++;                  counter++;                  counter++;
Release(l2, N1);            Release(l3, N2);            Release(l5, N3);

Acquire(l2, N1);
counter++;
Release(l2, N1);
```

Fig. 9: The client code for verifying and optimizing HMCS with VSync

possible on Arm. On IMM it can be fixed like indicated in Figs. 3(c) and 3(d) by adding $\mathbf{F_{acq}}$ at Event j.

4 Verifying and Optimizing HMCS with VSync

Vsync [15] is a fully automated verification and optimization tool that accepts (bounded) concurrent C/C++ programs as input. In its verification mode, it exhaustively enumerates all the possible executions of the input program following the GenMC model checking algorithm [12], and checks that these are terminating, memory-safe, and satisfy all user-supplied assertions. In its optimization mode, it uses an iterative algorithm to find minimal barrier placements that ensure program correctness (i.e., successful verification).

Like all model checkers, VSync does not verify locks abstractly: one must provide client code that uses the lock appropriately. A reasonable client must visit all functions and paths of a lock. For example, if in our client we configured HMCS with maximum `depth` $= 1$, the verification would cover only `AcqReal<1>` and `RelReal<1>`, and if we created only one thread then we would miss all concurrency bugs.

Verification time is generally super-exponential in the number of threads and acquire/release calls. We thus need to find the minimum number of threads with which we can still generate all bug-prone execution graphs. In general, finding this number is an open problem. We simply choose the maximum number of threads for which verification time is within reason. We experimentally justify this bound by adding an additional thread and observe that no additional bugs are found by VSync.

Our client code is shown in Fig. 9: it uses three threads, maximum `DEPTH` $= 3$, and thresholds `THRESHOLD[`d`]` $= 2$. We choose this maximum `DEPTH` $= 3$ because it covers the case where a lock (at depth $d = 2$) has both children and a parent. We assign threads to NUMA nodes as in Fig. 6. Each thread in our client acquires the lock, increments a shared variable, and then releases the lock. One thread (T_1) repeats this twice. This way we cover the case of a thread entering the critical section twice. With this setup, verification with VSync takes 10 seconds. Adding

a fourth thread T_4 on NUMA 0 (respectively, NUMA 3) increases verification time to 1300 seconds (respectively, 2800 seconds).

To verify mutual exclusion, we assert that our shared counter has the expected value after all threads are done (`assert(counter==4))`. If any execution graph of the client violates this assertion or indicates a non-terminating run of the program (such as the graphs in Figs. 4, 5 and 8), VSync prints that graph in text form. We note that debugging such graphs is non-trivial.

A simpler and more elegant way to use VSync is to implement the lock with only sequentially consistent memory operations (without fences). This ensures that there will be no bugs related to the consistency model. VSync then optimizes these barriers and reports to the user which barriers can be relaxed and/or removed. With our client code with three threads, optimization takes one second. With four threads, optimization takes less than 100 seconds.

5 Performance Evaluation

We evaluated HMCS on our Arm server and studied implementation choices that we expect are affecting performance. In particular, we tackle the following questions:

- Do the Large System Extensions (LSE) of Armv8.1 bring the promised performance improvements?
- Is there a performance penalty of unnecessary barriers, e.g., those introduced by VSync when optimizing for IMM rather than Arm?
- Do implicit barriers provide better performance than fences?

5.1 Experimental Setup

Environment. We ran the experiments on a Huawei TaiShan 200 (Model 2280) [1] with two HiSilicon Kunpeng 920-4826 processors [2] (2 packages), each of them with 48 Armv8.2 64-bit cores organized in 2 NUMA nodes and running at 2.6 GHz. The experiments reported in this section were conducted on openEuler 20.09 [3]. We reproduced similar results on Ubuntu 18.04 LTS.

Benchmarks. We conducted userspace experiments with LevelDB (`readrandom` benchmark) [7] and with a custom microbenchmark. In the microbenchmark, each thread repeatedly acquires a `pthread_mutex_lock`, increments a shared counter (causing a cache miss), and releases the mutex. In each experiment, we vary the number of threads and the lock implementation. We interpose calls to `pthread` functions with `LD_PRELOAD` in order to replace the lock implementation without modifying the benchmarks — in a similar fashion as [10]. We run each experiment for 3 seconds, repeat the experiment 10 times, and report the median throughput (number of iterations per second). We pin threads to cores from core 95 downwards, always keeping core 0 free to serve other OS tasks.

Lock variants. We compare the following variants of the HMCS-lock in regard to barrier/fence placements:

- hmcs-arm with a minimal set of fences required for Arm,
- hmcs-imm with a minimal set of fences required for IMM,
- hmcs-sc in which all racy accesses use sequentially consistent implicit barriers,
- hmcs-vsync with VSync-optimized implicit barriers, and
- hmcs-amo with optimized barriers on `CAS` and `SWAP` in hmcs-vsync.

We use the mcs lock with optimized barriers as a baseline. To avoid false sharing and ensure reliable results, we also cache-align and pad the shared data structures (`QNode` and `HNode`). All locks are implemented using C11 atomics (`stdatomic.h`).

5.2 Experimental Results for Low Contention

We start by exploring the performance of the HMCS variants with our microbenchmark running a single thread (see Fig. 10).

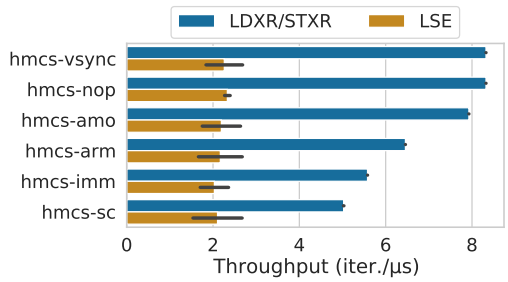


Fig. 10: Low contention scenario: Single-threaded microbenchmark with several HMCS variants compiled with and without LSE instructions.

LSE versus `LDXR/STXR`. HMCS variants that employ LSE instructions perform poorly in comparison to those that employ the conventional `LDXR`/`STXR` pair. The current hardware implementation of LSE degrades the performance of all HMCS variants. For example, in the case of hmcs-vsync, the throughput of the LSE version is 27% of the `LDXR`/`STXR` throughput. We also observe that LSE implementations tend to have a higher variance than the `LDXR`/`STXR` implementations (see standard deviation reported on Fig. 10).

Due to the importance of the single-thread scenario, we only consider the implementations with `LDXR`/`STXR` in the remainder of the evaluation.

Performance penalty of no optimization. The performance of hmcs-sc shows that exclusively employing sequentially consistent implicit barriers incurs a considerable cost under low contention. As we will see below, the performance of hmcs-sc is comparable to the other variants under high contention.

Performance penalty of targeting IMM. We observe that hmcs-arm has 16% higher throughput than hmcs-imm, implying that the additional fences required by IMM impact the performance negatively. In contrast, the additional implicit barriers in hmcs-vsync do not reduce throughput compared to hmcs-amo, which has been manually optimized for Arm. This suggests that the performance penalty of using IMM as the verification target depends on the type of barriers and not simply on the number of additional barriers required by IMM.

Implicit barriers versus fences. Automatically-selected implicit barriers perform better than fences: hmcs-vsync shows 49% higher throughput than hmcs-imm, and 28% higher throughput than hmcs-arm. Note that replacing fences with implicit barriers reduces the code length, which in turn can shorten single-threaded runs and improve the instruction cache usage. To validate that the shorter code length is not the source of the improved performance of hmcs-vsync over hmcs-arm, we create a variant based on hmcs-vsync, in which we introduce a `NOP` instruction for every removed fence (`NOP` and fences have the same length in Arm); we call this variant hmcs-nop. Figure 10 shows a negligible difference between hmcs-vsync and hmcs-nop, corroborating the claim that implicit barriers improve performance for *single* threaded code [13]. Nevertheless, whether implicit barriers or fences perform better for *multiple* threads may depend on the benchmark, as we will see below.

The reason for this discrepancy is not clear; besides micro-architectural implementation details, a possible reason may lie in the weak consistency model of Arm itself. For the correctness of the HMCS lock, the order between specific loads and subsequent memory operations needs to be enforced. On the Kunpeng 920 server, these loads can be implemented either as a load with a trailing `DMB LD` instruction (acquire fence), or as `LDAR`/`LDAXR` load instructions with implicit acquire barriers. Both are unsatisfactory. The `DMB LD` instruction needlessly orders *all previous* loads with subsequent operations. The `LDAR`/`LDAXR` instructions needlessly order all previous stores with implicit release barriers with that load. These non-comparable unnecessary ordering constraints might be the reason both implementation choices are sometimes the better choice. Armv8.3 (not supported on Kunpeng 920) introduces the `LDAPR` load instruction, which only introduces the necessary order. Perhaps the comparison between implicit barriers and fences would be more clear-cut with this instruction.

5.3 Experimental Results for High Contention

We now explore higher contention scenarios with our microbenchmark and with LevelDB benchmark. In the following experiments, we consider hmcs-arm, hmcs-vsync, hmcs-sc, and mcs.

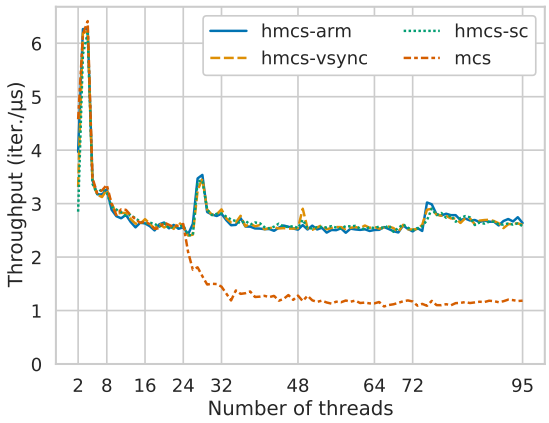


Fig. 11: Microbenchmark with 2 to 95 threads and different lock implementations.

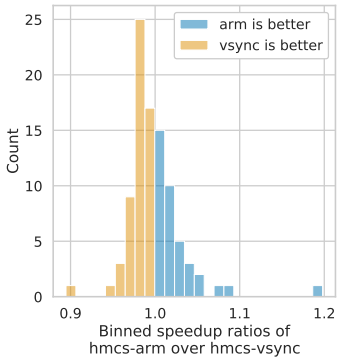


Fig. 12: Speedup histogram of hmcs-arm over hmcs-vsync for the microbenchmark with 1 to 95 threads.

Figure 11 shows the performance of our benchmark running with 2 to 95 threads. (Single-threaded runs were evaluated in Section 5.2, and one core is left free to handle interrupts, which are otherwise a source of noise.) We assign threads to cores sequentially.

After filling a complete NUMA node (with 24 threads), the performance of mcs drops considerably; for example at 95 threads, mcs throughput is about 45% of hmcs-vsync throughput. The performance spike with 4 threads is due to the higher cache locality achieved when threads share the same L3 cache region. Kunpeng 920 processors split the L3 cache in regions shared by groups of 4 cores. The spike with 28 threads is caused by the interplay of the HMCS policy to keep the lock in the NUMA node and the fact that 4 cores of the second NUMA node share the same L3 cache region. HMCS enforces that both NUMA nodes have the same share of the lock with a user-configured threshold (see Section 2.1). Therefore, the first NUMA node executes half of the benchmark iterations with 24 cores, whereas the second NUMA node executes the other half with 4 threads and few L3 cache misses, improving the overall throughput. The spike repeats at lower intensities when the other NUMA nodes only use 4 cores.

The different HMCS variants perform in most configurations less than 10% apart. Figure 12 shows the histogram of speedups of hmcs-arm over hmcs-vsync for 2 to 95 threads. The single case around 0.89 is with 49 threads, where hmcs-arm is slower than hmcs-vsync. The single case around 1.20 is with 2 threads, where hmcs-arm is faster than hmcs-vsync: this is caused by the slowpath of MCS lock release, which is triggered more often with hmcs-vsync and 2 threads. For the other cases, we observe that hmcs-arm tends to be slightly slower than hmcs-vsync, but the difference is below 8%.

Figure 13 shows the performance of the LevelDB benchmark running 1 to 95 threads. The benchmark contains parallel work and can scale up to around 8

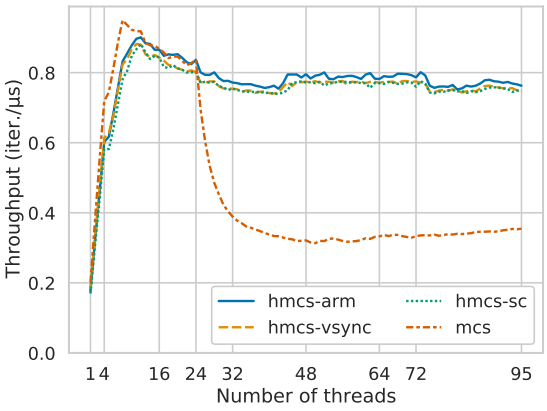


Fig. 13: LevelDB benchmark with 1 to 95 threads and different lock implementations.

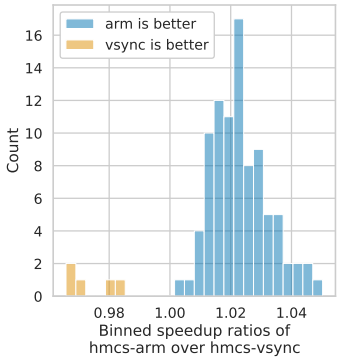


Fig. 14: Speedup histogram of hmcs-arm over hmcs-vsync for the LevelDB benchmark with 1 to 95 threads.

threads. Up to 9 threads, mcs performs up to 20% faster than hmcs-vsync, but continuously degrades its throughput when more than 24 threads are running (or more than one NUMA node is used in the application). For example at 95 threads, the mcs throughput is 47% of hmcs-vsync throughput. Between 10 and 24 threads, hmcs-vsync and mcs are at most 6% apart.

Figure 14 shows again the histogram of speedups of hmcs-arm over hmcs-vsync for 1 to 95 threads. In the range from 1 to 5 threads, hmcs-vsync performs up to 4% faster than hmcs-arm. With 6 threads or more, hmcs-arm performs up to 5% faster than hmcs-vsync.

Finally, hmcs-vsync performs up to 8% faster than hmcs-sc in the range from 1 to 10 threads. With 11 threads or more, the hmcs-vsync throughput is between 0.99 to 1.03 times the hmcs-sc throughput.

6 Discussion

Already with sequentially consistent barriers, the NUMA-aware HMCS lock considerably outperforms the MCS lock at high levels of contention. At these levels, the performance impact of barrier optimization is negligible. On the other hand, incorrect optimizations can lead to heisenbugs. For this reason, we recommend simply using sequentially consistent barriers on all racy accesses, and not worrying about weak consistency. In cases of low contention, however, barrier optimizations can show substantial performance improvements. In these cases, the automatic and formally verified optimizations by VSync outperform manual optimizations (both our own and the repaired fences from the literature). This shows that barrier optimization, if desired, should be left to the machine.

References

1. https://e.huawei.com/uk/products/servers/taishan-server/taishan-2280-v2
2. https://en.wikichip.org/wiki/hisilicon/kunpeng/920-6426
3. https://openeuler.org
4. Amazon Web Services: AWS Graviton Processor – Enabling the best price performance in Amazon EC2 (2020), https://aws.amazon.com/ec2/graviton
5. Chabbi, M., Amer, A., Wen, S., Liu, X.: An efficient abortable-locking protocol for multi-level NUMA systems. SIGPLAN Not. **52**(8), 61–74 (Jan 2017). https://doi.org/10.1145/3155284.3018768
6. Chabbi, M., Fagan, M.W., Mellor-Crummey, J.M.: High performance locks for multi-level NUMA systems. In: PPoPP 2015. pp. 215–226. ACM, New York, USA (2015). https://doi.org/10.1145/2688500.2688503
7. Dean, J., Ghemawat, S.: Leveldb. https://github.com/google/leveldb (2021)
8. Defilippi, J.: Introducing AMBA 5 CHI protocol enhancements (2017), https://community.arm.com/developer/ip-products/system/b/soc-design-blog/posts/introducing-new-amba-5-chi-protocol-enhancements
9. Dice, D., Kogan, A.: Compact NUMA-aware locks. In: EuroSys 2019. ACM, New York, USA (2019). https://doi.org/10.1145/3302424.3303984
10. Guiroux, H., Lachaize, R., Quéma, V.: Multicore locks: The case is not closed yet. In: USENIX Annual Technical Conference. pp. 649–662 (2016)
11. Huawei: Huawei unveils industry's highest-performance ARM-based CPU (Jan 2019), https://www.huawei.com/en/news/2019/1/huawei-unveils-highest-performance-arm-based-cpu
12. Kokologiannakis, M., Raad, A., Vafeiadis, V.: Model checking for weakly consistent libraries. In: PLDI 2019. pp. 96–110. ACM, New York, USA (2019). https://doi.org/10.1145/3314221.3314609
13. Liu, N., Zang, B., Chen, H.: No barrier in the road: A comprehensive study and optimization of ARM barriers. In: PPoPP 2020. pp. 348–361. ACM, New York, USA (2020). https://doi.org/10.1145/3332466.3374535
14. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst. **9**(1), 21–65 (Feb 1991). https://doi.org/10.1145/103727.103729
15. Oberhauser, J., de Lima Chehab, R.L., Behrens, D., Fu, M., Paolillo, A., Oberhauser, L., Bhat, K., Wen, Y., Chen, H., Kim, J., Vafeiadis, V.: VSync: Push-button verification and optimization for synchronization primitives on weak memory models. In: ASPLOS 2021. ACM, New York, USA (2021). https://doi.org/10.1145/3445814.3446748
16. Podkopaev, A., Lahav, O., Vafeiadis, V.: Bridging the gap between programming languages and hardware weak memory models. Proc. ACM Program. Lang. **3**(POPL) (Jan 2019). https://doi.org/10.1145/3290382
17. Pulte, C., Flur, S., Deacon, W., French, J., Sarkar, S., Sewell, P.: Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8. Proc. ACM Program. Lang. **2**(POPL) (Dec 2017). https://doi.org/10.1145/3158107

A Arm vs. IMM Consistency Model

$$
\begin{aligned}
\mathit{obs} &\supseteq \mathbf{rfe} \cup \mathbf{moe} && (1)\\
\mathit{dob} &\supseteq \mathbf{ctrl}; [W] && (2)\\
\mathit{bob} &\supseteq \mathbf{po}; [\mathbf{F_{rel}}]; \mathbf{po} && (3)\\
\mathit{lob} &\supseteq \mathit{dob} \cup \mathit{bob} && (4)\\
\mathbf{ob} &\supseteq \mathit{obs} \cup \mathit{lob} \cup \mathbf{ob}; \mathbf{ob} && (5)\\
\mathbf{ob} &\text{ is irreflexive} && (6)
\end{aligned}
$$

Fig. 15: A Subset of the Arm Consistency Model. The key derived relation is **o**rdered-**b**efore (**ob**), which is irreflexive in consistent graphs.

$$
\begin{aligned}
F &\supseteq \mathbf{F_{rel}} \cup \mathbf{F_{acq}} && (7)\\
\mathit{deps} &\supseteq \mathbf{ctrl} && (8)\\
\mathit{ppo} &\supseteq [\mathbf{R}]; \mathit{deps}; [\mathbf{W}] && (9)\\
\mathit{bob} &\supseteq [F]; \mathbf{po} \cup \mathbf{po}; [F] && (10)\\
\mathbf{ar} &\supseteq \mathbf{rfe} \cup \mathit{bob} \cup \mathit{ppo} && (11)\\
\mathbf{ar} &\text{ is acyclic} && (12)\\
\mathit{release} &\supseteq [\mathbf{F_{rel}}]; \mathbf{po} && (13)\\
\mathbf{sw} &\supseteq \mathit{release}; \mathbf{rfe}; \mathbf{po}; [\mathbf{F_{acq}}] && (14)\\
\mathbf{hb} &\supseteq \mathbf{po} \cup \mathbf{sw} \cup \mathbf{hb}; \mathbf{hb} && (15)\\
\mathbf{eco} &\supseteq \mathbf{rfe} \cup \mathbf{moe} && (16)\\
\mathbf{hb}; \mathbf{eco} &\text{ is irreflexive} && (17)
\end{aligned}
$$

Fig. 16: A Subset of the IMM Consistency Model. Key relations are the **a**cyclic **r**elation (**ar**) which is acyclic in consistent graphs, as well as **s**ynchronizes-**w**ith (**sw**), **e**xtended **c**oherence **o**rder (**eco**), and **h**appens-**b**efore (**hb**), where **hb**; **eco** is irreflexive in consistent graphs.

A standard way to define weak consistency models is through execution graphs. **Nodes** in these graphs represent events such as reads and writes, and **edges** specify various relations between these events, e.g., the order in which reads and writes to the same location are committed. Memory models are defined by a) the edges that exist in the graph and b) restrictions on these edges. For brevity, we introduce only the event and edge types of Arm and IMM that are relevant to the bugs we mention in this paper. We consider write events $\mathbf{W}_X(\mathit{loc}, \mathit{val})$, read events $\mathbf{R}_X(\mathit{loc}, \mathit{val})$, and fence events $\mathbf{F}_X$, where $X \in \{\, \mathbf{sc}, \mathbf{acq}, \mathbf{rel}, \mathbf{rlx} \,\}$ is the so called **mode** of the event, *loc* is the shared memory location on which the event operates, and *val* is the value written or read in the event. The mode denotes the type of memory barrier (if any) represented by the event: **rlx** indicates that no barrier is present, **acq** represents acquire, **rel** release, and **sc** sequentially consistent barriers. Mode **rlx** is the default mode and omitted.

We consider the following types of fundamental edges:

- **rfe** (**r**ead **f**rom **e**xternal) edges $\mathbf{W}_X(x,a) \xrightarrow{\text{rfe}} \mathbf{R}_Y(x,a)$ connect a write event of a thread to a read event of another thread that reads from it.
- **moe** (**m**odification **o**rder **e**xternal) edges $\mathbf{W}_X(x,a) \xrightarrow{\text{moe}} \mathbf{W}_Y(x,b)$ connect write events (writing to the same location) of different threads indicating the order in which they were committed.
- **po** (**p**rogram **o**rder) edges connect events of the same thread in the order in which they are issued by the program.
- **ctrl** (**control** dependency) edges connect a read $\mathbf{R}_X(x,a)$ that influences a condition (e.g., if- or while-condition) evaluation to every event of the same thread that is issued after the condition.
- event-type self-loops $e \xrightarrow{[E]} e$ for event type $E \in \{\, \mathbf{R}, \mathbf{W}, \mathbf{F_{rel}}, \mathbf{F_{acq}} \,\}$ connect every event e of type E to itself.

Other edges are derived from these fundamental edges according to the rules of the consistency model (Figs. 15 and 16). For instance, the edge a $\xrightarrow{\text{moe}}$ e in Fig. 3(b) implies an **eco** edge a $\xrightarrow{\text{eco}}$ e on IMM (with Eq. (16)). Such derived rules are often defined with the composition operator '**;**', which for arbitrary edge types R and S is defined by

$$a \xrightarrow{R;S} c \iff a \xrightarrow{R} b \xrightarrow{S} c$$

The meaning of barriers is defined by the derived edges they imply; for example, the meaning of $\mathbf{F_{rel}}$ (which maps to the full `DMB.ISH` fence) on Arm is defined through the **ob** edge it implies between preceding and subsequent operations (with Eqs. (3) to (5)).

In Figs. 15 and 16 we have collected the rules of IMM and Arm consistency that are relevant to our discussion. In [16] it is shown that Arm consistency implies IMM consistency; thus any bug on Arm is also present on IMM, and verification on IMM implies correctness on Arm. The converse is not true, and bugs on IMM are not always bugs on Arm. Indeed, some of the bugs identified by VSync on the HMCS lock on IMM are not bugs on Arm. The key difference relevant to these bugs is that **moe** edges imply an **ob** edge on Arm, but do not imply an **ar** edge on IMM. Thus they contribute to **ob** cycles but not to **ar** cycles.

We illustrate the implications at hand of the execution graphs in Fig. 3. In Fig. 3(a), we have an **rfe** edge from Event e to Event a; in Fig. 3(b), we instead have an **moe** edge from Event e to Event a. Other than those events and the edge between them, the graphs are the same. Thus in both graphs, the following imply **ob** edges:

- a $\xrightarrow{\text{po}}$ b $\xrightarrow{[\mathbf{F_{rel}}]}$ b $\xrightarrow{\text{po}}$ c (Eqs. (3) to (5))
- c $\xrightarrow{\text{rfe}}$ d (Eqs. (1) and (5))
- d $\xrightarrow{\text{ctrl}}$ e $\xrightarrow{[W]}$ e (Eqs. (2), (4) and (5))

The only edge missing for an **ob** cycle is e $\xrightarrow{\text{ob}}$ a. This edge is implied by the e $\xrightarrow{\text{rfe}}$ a edge in Fig. 3(a) and the e $\xrightarrow{\text{moe}}$ a edge in Fig. 3(b) (with Eqs. (1) and (5)). Note that due to transitivity (Eq. (5)) the cycle a $\xrightarrow{\text{ob}}$ ... $\xrightarrow{\text{ob}}$ a implies a reflexive edge a $\xrightarrow{\text{ob}}$ a, which contradicts the irreflexivity of **ob** (Eq. (6)). Thus both graphs are inconsistent on Arm.

On IMM, the following imply **ar**-edges:

- a $\xrightarrow{\text{po}}$ b $\xrightarrow{[\mathbf{F_{rel}}]}$ b and b $\xrightarrow{[\mathbf{F_{rel}}]}$ b $\xrightarrow{\text{po}}$ c (Eqs. (7), (10) and (11))
- c $\xrightarrow{\text{rfe}}$ d (Eq. (11))
- d $\xrightarrow{[\text{R}]}$ d $\xrightarrow{\text{ctrl}}$ e $\xrightarrow{[\text{W}]}$ e (Eqs. (8), (9) and (11))

Analogous to before, only an e $\xrightarrow{\text{ar}}$ a is missing for an **ar** cycle. In Fig. 3(a) this edge is implied by the e $\xrightarrow{\text{rfe}}$ a edge with Eq. (11), and this graph is inconsistent on IMM. But in Fig. 3(b), the **moe** edge does not contribute an **ar** edge. Indeed, there is no **ar** cycle in Fig. 3(b), which is consistent on IMM. Unfortunately, two of the bugs detected by VSync on IMM appear only in graphs that look like Fig. 3(b). These bugs therefore only appear on IMM, but can not appear on Arm.

We proceed to discuss how to fix these bugs on IMM. Consider the third graph (see Fig. 3(c)) which is almost identical to the second (see Fig. 3(b)). We only added a $\mathbf{F_{acq}}$ fence between the d and f. Adding this fence does not eliminate the **ob**-cycle we inferred previously, and this graph is also inconsistent with Arm. On IMM we derive the following edges:

- b $\xrightarrow{[\mathbf{F_{rel}}]}$ b $\xrightarrow{\text{po}}$ c $\xrightarrow{\text{rfe}}$ d $\xrightarrow{\text{po}}$ e $\xrightarrow{[\mathbf{F_{acq}}]}$ e thus b $\xrightarrow{\text{sw}}$ e (Eqs. (13) and (14))
- a $\xrightarrow{\text{hb}}$ b, b $\xrightarrow{\text{hb}}$ e and e $\xrightarrow{\text{hb}}$ f thus a $\xrightarrow{\text{hb}}$ f (Eq. (15))
- f $\xrightarrow{\text{eco}}$ a (Eq. (16))

As shown in (Fig. 3(d)) we end up with a $\xrightarrow{\text{hb}}$ f $\xrightarrow{\text{eco}}$ a and thus a $\xrightarrow{\text{hb;eco}}$ a. But the **hb**;**eco** relation is irreflexive (Eq. (17)). We conclude that this graph is inconsistent with IMM. In other words, due to the $\mathbf{F_{acq}}$ fence the execution with the bug cannot occur on IMM.

B Optimizing Barriers on Atomic Operations

The implicit **sc** barriers on `CAS` and `SWAP` in Fig. 2 are not optimal. VSync reports that they are already too strong for IMM, and indeed they can be optimized further for Arm. The exact optimization depends on the variant. Manual analysis shows that when using fences, all barriers on the atomic operations can be removed. When using implicit barriers, release barriers on Lines 17 and 30 are needed to avoid non-termination (with similar bugs as those in Section 3.1) and acquire and release barriers are needed on Line 17 resp. Line 55 to ensure that operations in the critical section can not leak out of the lock (resulting in loss of mutual exclusion). The resulting barriers are shown in Table 1. That

	`SWAP` [Line 17]	`SWAP` [Line 30]	`CAS` [Line 55]
Fences	-	-	-
Implicit	**sc**	**rel**	**rel**
Implicit (LSE)	**rel** ; $\mathbf{F_{acq}}$	**rel**	**rel**

Table 1: Possible optimizations on Arm for atomic operations when using fences or implicit barriers.

table also shows a variant that may be more optimal when using interlocked LSE instructions. Unlike load/store-exclusive pairs, on which **sc** implicit barriers do not act like a full barrier (see discussion in Section 3.1), LSE interlocked operations have been strengthened in a recent change to Arm specifications to provide the same semantics for **sc** implicit barriers as a `DMB.ISH` (see Eq. (10) from Page 17) through the rule

$$bob \supseteq \mathbf{po}; ([\mathbf{A}]; amo; [\mathbf{L}]); \mathbf{po}$$

where *amo* relates a the read event of an **a**tomic **m**emory **o**peration (such as `SWP`) to its write event, and [**A**] and [**L**] are event-type self-loops for acquire resp. release events. This contrasts the earlier definition in [17], in which LSE instructions provide the same ordering guarantees as load/store-exclusive pairs.

However, this stronger ordering is not necessary for the HMCS lock, and thus we optimize barriers further by relegating the acquire barrier to a trailing fence. This variant is what is denoted by hmcs-amo in Section 5. As demonstrated in Fig. 10 on Page 12, this optimization does not currently improve performance compared to hmcs-vsync (which uses **sc** barriers on atomic operations). Perhaps if LSE operations become more efficient for low-contention cases in the future, these optimizations will become more interesting.

For the sake of completeness we also implement a variant hmcs-armamo which applies the optimization to hmcs-arm, i.e., in which as described in Table 1 all implicit barriers on atomic operations are removed. Performance results (without LSE) are shown in Figs. 17 and 18. While minor improvements can be measured in the microbenchmark, these improvements also do not translate to the larger benchmark.

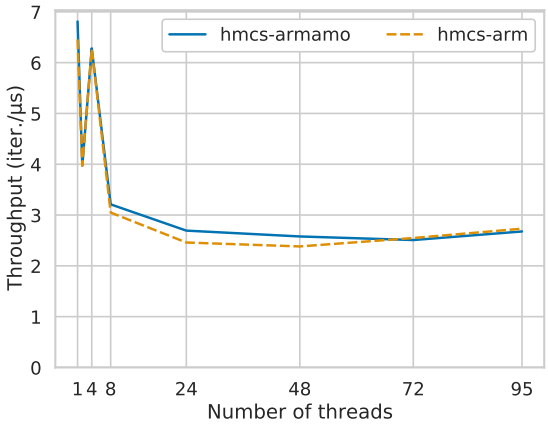


Fig. 17: Performance of AMO-optimizations with fences on microbenchmark

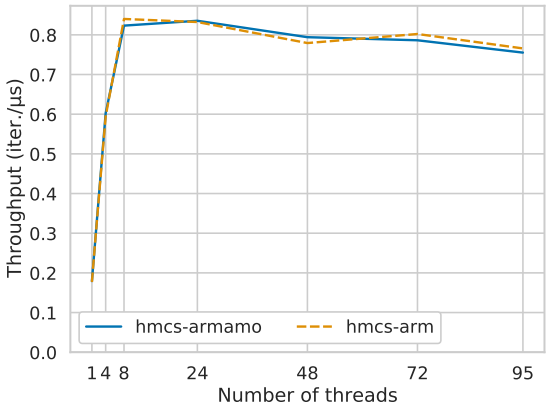


Fig. 18: Performance of AMO-optimizations with fences on LevelDB