AtoMig: Automatically Migrating Millions Lines of Code from TSO to WMM

Martin Beck
Huawei Dresden Research Center,
Huawei Central Software Institute
Dresden, Germany

Koustubha Bhat
Huawei Dresden Research Center,
Huawei Central Software Institute
Dresden, Germany

Lazar Stričević
Huawei Dresden Research Center,
Huawei Central Software Institute
Dresden, Germany

Geng Chen
Huawei Fundamental Software Innovation Lab, Huawei Central Software Institute
Shenzhen, China

Diogo Behrens
Huawei Dresden Research Center,
Huawei Central Software Institute
Dresden, Germany

Ming Fu[*]
Huawei Dresden Research Center,
Huawei Central Software Institute
Dresden, Germany

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

Haibo Chen
Huawei Central Software Institute
Shenzhen, China
Shanghai Jiao Tong University
Shanghai, China

Hermann Härtig
Technische Universität Dresden
Dresden, Germany

ABSTRACT

CPUs with weak memory-consistency models (WMMs), such as Arm and RISC-V, are rapidly increasing their market share. Porting legacy x86 applications to such CPUs requires introducing extra synchronization to prevent WMM-related concurrency bugs—a task often left to human experts.

Given the rarity of such experts and the enormous size of legacy applications, we develop AtoMig, an effective, fully automated tool for porting large, real-world applications to WMM CPU architectures. AtoMig detects shared memory access patterns with novel static analysis strategies and performs program transformations to properly protect them from WMM effects. In the absence of sufficiently scalable verification methods, AtoMig shows practicality of focusing on code patterns more prone to WMM faults, trading off completeness for scalability.

We validate the correctness of AtoMig's transformations on several small concurrent benchmarks via model checking. We demonstrate the scalability and performance of our approach by applying AtoMig to popular real-world large code bases with up to millions of lines of code, viz., MariaDB, PostgreSQL, SQLite, LevelDB, and Memcached. As part of this work, we also found a WMM bug in MariaDB, which AtoMig fixes automatically.

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

CCS CONCEPTS

• **Software and its engineering** → **Consistency**; **Automated static analysis**; • **Theory of computation** → **Concurrency**.

KEYWORDS

memory consistency models, parallelism and concurrency, static analysis, sustainability

ACM Reference Format:
Martin Beck, Koustubha Bhat, Lazar Stričević, Geng Chen, Diogo Behrens, Ming Fu, Viktor Vafeiadis, Haibo Chen, and Hermann Härtig. 2023. AtoMig: Automatically Migrating Millions Lines of Code from TSO to WMM. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 13 pages.

1 INTRODUCTION

CPU families with weak memory-consistency models (WMMs) like Arm and RISC-V are becoming increasingly pervasive [20, 33]. Besides mobile devices, Arm cores are now powering compute instances in the AWS cloud [19], supercomputers [59], high-performance servers [39], Microsoft Surface laptops [67], and several Apple end-user devices [68]. Although not as prominent as Arm, RISC-V market share is also steadily increasing and is projected to reach 6% by 2025 [33].

Given the pervasiveness of WMM CPUs, an important need has risen in the industry for an easy and automated way of porting existing applications to them.

Our motivation for this work largely stems from a big industry software project that was developed and ran for years on x86 machines. Much later, a business need arose to run that software on Arm-based servers. The software had a critical dependency on DPDK [29] library. This library was so deeply integrated within the software that it was no longer straightforward to replace it with its

```
        int flag, msg = 0;

// reader              || // writer
while(flag == 0);      || msg = 1;
assert(msg == 1);      || flag = 1;
```

Figure 1: Message passing example.

newer, correctly ported version for Arm servers. The engineers resorted to manually port the software. Consequently, several WMM bugs surfaced when this was deployed. Despite several person-years of fixing efforts to finally make those WMM bugs disappear, it still turned out that they were incorrectly fixed.

Automatically porting and fixing such WMM bugs, however, is not easy. Legacy concurrent applications were neither designed nor tested for WMMs, but for an older, fairly strong memory model: x86-TSO [62]. WMMs allow many more reorderings among memory operations than TSO (e.g., stores to different memory addresses). These additional reorderings adversely affect thread synchronization, while the rest of the code base typically works out of the box, with appropriate instruction translation to the target architecture. For example, without extra care, the standard "message passing" pattern (as in Figure 1), where a writer thread initializes and publishes an object, and another thread reads it, can lead to reading corrupt data. To prevent such erroneous behaviors, one has to enforce some ordering among memory operations. To this end, WMMs provide *barriers*, which are either stand-alone explicit barriers, also called fences (e.g., DMB on Arm) or implicit barriers attached to memory operations (e.g., LDAR and STLR on Arm).

Placing such barriers inside the code is non-trivial. On one hand, they are necessary for reestablishing the orderings required by the algorithm. On the other hand, unnecessary or overly-constrained barriers degrade the performance of the complete system because concurrent code often lies on the critical path. For example, a single unnecessary barrier in the spinlock of Linux reduced the performance of the whole kernel by 4% [1].

Currently, the placement of barriers is typically handled by human experts, who 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 [26, 49–51, 70]. Unfortunately, this is an error-prone task, even for experts. For example, the optimization of the barriers in Linux's qspinlock introduced a bug [49] that remained unfixed for three years [26].

More importantly, manual porting cannot scale for large code bases, such as databases and web servers, due to the lack of sufficiently many experts who understand both WMMs and the system in detail. A telling story revolves around a WMM bug in MariaDB [5], a large open-source database system. The errors resulting from that bug were recognized, but misunderstood. Instead of fixing the WMM bug, the elements of a data structure were reordered and some `volatile` annotations removed, which led an optimization pass of the compiler to combine two loads of successive members of that data structure into a single instruction, thereby hiding the bug. On non-optimized builds, however, the bug still frequently occurred. Since no test case existed for this scenario, the non-fix was accepted as a fix for the WMM bug.

The state of the art in automated barrier placement is also not yet suitable for the task. Approaches based on model checking [12, 14, 17, 25, 41–44] do not scale because of the well-known state explosion problem. Approaches based on overapproximation [16, 60] require expensive inter-procedural static analyses and/or yield poor performance. Testing-based approaches [61] require substantial human input in terms of curated test harnesses.

Our Solution. In response, we develop AtoMig, the first highly-performant static approach for porting real-world large-scale applications from x86-TSO to WMMs. We demonstrate AtoMig's scalability and performance by applying it to popular real-world large code bases with millions of lines of code, viz., MariaDB, PostgreSQL [10], SQLite [11], LevelDB [8] and Memcached [9]. It successfully ports them to Arm with a performance overhead of 1.8% within minutes, which is roughly the same time it takes to compile the applications.

The key challenge in porting a program from x86 to WMMs is to detect its set of concurrent memory accesses that are not adequately protected by locks and/or other explicit synchronization constructs. One cannot rely purely on annotations like `volatile` declarations in C because programmers often forget to annotate all concurrent accesses. Once this set is detected, we can insert appropriate barriers to prevent reorderings among these accesses and between them and other memory accesses, thereby implementing TSO semantics. Where possible, our approach uses implicit instead of explicit barriers, because implicit barriers tend to be much faster [48].

Our key idea to find those missing accesses is to look for code patterns that (1) correlate highly with the presence of concurrent synchronization accesses and (2) can be detected efficiently by intra-procedural analysis with very few false positives. The latter is crucial to achieve scalability and high performance. We thus focus our attention to spinloops, which not only can be detected efficiently, but are also the places where WMM bugs are the most likely to occur in real-world systems given the fairly low observed probability of WMM behaviors in general [18]. From those detected spinloops, AtoMig employs a simple scalable alias analysis to find all other accesses to the same memory locations, thereby repairing also the code in concurrency patterns that are difficult to detect effectively.

In the absence of scalable verification tools, we present a practical trade-off in the spectrum between poorly efficient but formally correct porting and, highly efficient porting with limited correctness guarantees. AtoMig, being heuristic-based, provides no formal correctness guarantees (in fact, similar to widely applied manual porting for large applications). We validate AtoMig's correctness on smaller concurrent benchmarks which are within the reach of automatic verification techniques, including widely used data structures from the Concurrency Kit library [15]. Moreover, AtoMig was instrumental in exposing and (automatically) fixing a WMM bug in a critical hash-table implementation in the MariaDB database, with the fix merged into its current code base [4].

In summary, we claim that AtoMig is the first *practical* software-based approach to porting large code bases from x86-TSO to WMM because

(1) it scales to millions of lines of code,
(2) achieves a very low (1.8%) performance overhead,

(3) does not require any user input, and
(4) automatically fixes WMM bugs that matter.

Its design is based on a novel use of spinloops as entry points for scalably detecting shared memory accesses and protecting them from undesirable WMM effects.

2 BACKGROUND

2.1 Weak Memory Models

Memory models define the semantics of memory accesses in multi-threaded programs at the level of hardware architectures like Intel x86, Power, or Armv8 or at the level of programming languages like C11 and Java. They define which accesses may be executed or have their effects propagated out of order, and what primitives — explicit or implicit barriers — exist to prevent such reorderings.

The strongest memory model is *sequential consistency* (SC) [47], which guarantees memory accesses to happen in program order. A slightly weaker model is *Total Store Order* (TSO) [62], which relaxes the store-to-load order, thereby allowing the weak "store buffering" behavior. TSO is commonly used to describe the Intel x86 memory order for pages whose attribute is set to "write-back" (the default mode for user-space applications).

Weak Memory Models (WMMs) like the ones of Power, Arm, RISC-V, C11, Java, etc., further relax the execution order between instructions, allowing CPUs to execute independent memory accesses completely out of order. This additional implementation flexibility translates into better performance, less energy consumption, but also more weak behaviors, especially in programs containing the "load buffering" and "message passing" patterns.

As Intel x86 was the prevalent architecture for several decades, most concurrent software is written for the TSO memory model, and it is very important to be able to port them to run correctly and efficiently on WMM machines.

2.2 The Porting Problem

Porting software written for the TSO memory model to a WMM architecture is sadly not simply a matter of recompiling the program. Doing so can easily lead to hard-to-spot weak memory model consistency bugs, since the additional reorderings allowed by WMMs can generate unintended program behaviors. To prevent such harmful behaviors, one has to introduce memory barriers and/or similar synchronization constructs at appropriate points.

Finding exactly where to introduce such barriers is a non-trivial task. Inserting too many barriers can incur a significant performance penalty, while not having enough barriers can lead to program errors. This has opened up a problem space with many proposed approaches over the last decades. In Table 1, we review the existing solutions for porting software from TSO to a WMM. We compare the solutions in terms of *Safety* i.e., final result is guaranteed to be safe; *Efficiency* i.e., final result has low performance overhead; *Scalability* i.e., number of lines of code that the method can be applied on; and *Practicality* i.e., whether it can easily be used on a variety of existing applications without requiring a lot of domain knowledge (client code, a curated test suite, etc.). We denote if the proposed solution (mostly) fulfills the property with a ✓, with a ✗ if it (mostly) does not fulfill the property and if that the proposed solution partly fulfills the property with a =.

Table 1: Comparison of Porting Approaches

Approach	Safe	Efficient	Scalable	Practical
Naïve	✓	✗	✓	✓
Hardware	✓	=	✓	=
Expert	=	✓	✗	✗
VSync [57]	✓	✓	✗	✗
Musketeer [16]	✓	=	=	✗
Lasagne [60]	✓	✗	✓	✗
TSan [61]	✗	=	=	✗
AtoMig	=	✓	✓	✓

Naïve Solution. The simplest solution is to make all memory accesses SC by using Arm's implicit SC barriers or by inserting a strong memory barrier between every pair of shared memory accesses. This solution fulfills our safety, scalability, and practicality requirements, but introduces significantly high runtime overhead. Among the choice of using implicit or explicit SC memory barriers, the former is observed to offer higher performance.

Hardware-Based Solutions. Some CPUs support multiple memory orderings. For example, the Armv8-based CPU cores on Apple M1 SoC support code execution in WMM or TSO modes. In combination with Rosetta 2 dynamic binary translation [7], TSO mode enables running legacy x86-64 macOS programs on the new Armv8 platform. The solution is safe and scalable. On the other hand, it is highly platform-specific and not all applications can be translated in this way [27] (e.g., AVX instructions are not supported [7]). Also, its performance is not ideal. To evaluate, we ran both x86 [32] and native [31] versions of Geekbench [30] for macOS on the same machine. The x86 version achieved about 22% lower Geekbench score.

Relying on Experts. Another (non-)solution is to rely on human experts to manually introduce any additional barriers in the code base, which is historically how most porting efforts have proceeded. While this approach can yield excellent performance, it is neither practical nor scalable: it requires experts who have a very good and detailed understanding of the target software and the memory model to be ported to.

Verification-Backed Approaches. Another approach, followed by projects like VSync [57], is to construct a candidate placement of memory barriers and to use verification tools (typically, model checkers) to prove the correctness of the barrier assignment, refining the assignment depending on the verification output. The major problem with this approach is scalability: model checkers examine all possible distinct program executions and cannot verify programs larger than a few thousand lines of code. A secondary problem is that of practicality in that these verification tools often require manual code curation or annotations before they can be applied.

Barrier Insertion Approaches. Another approach is to employ static analysis to detect programming patterns that require memory barriers to avoid weak behaviors on a given WMM architecture. Musketeer [16] does so by generating abstract event graphs of the possible executions of a program, applying axiomatic rules checking for

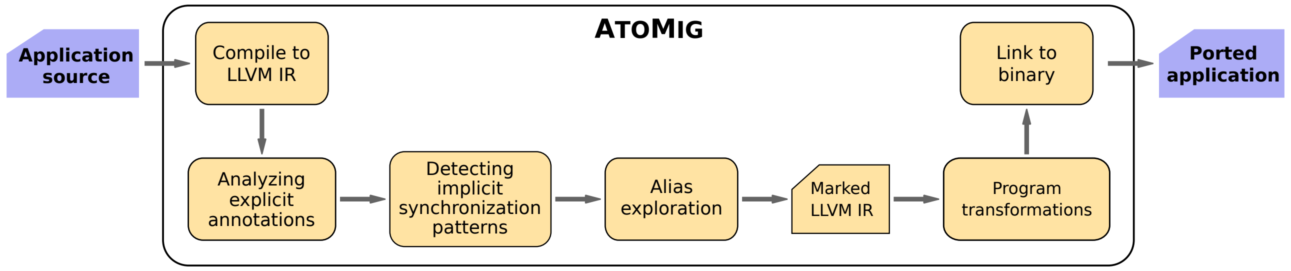

Figure 2: AtoMig's workflow

memory consistency bugs and inserting barriers to fix them. The main problems with this approach are its practicality and scalability: the analysis depends on client code coverage and hits a scalability bottleneck due to its heavy reliance on precise alias analysis.

Barrier Removal Approaches. Lasagne [60] is the converse approach. It starts by making the whole application SC and then detects patterns provably unrelated to synchronization to eliminate unnecessary barriers. While its analysis scales well, it often does not manage to remove many barriers and so it introduces a high performance overhead. The use of explicit memory barriers instead of implicit SC barriers also impacts performance. Further, it is built into the static binary translation tool mctoll [71], which comes with its own limitations. It only supports a subset of the x86 instruction set and needs function prototypes to be input for all called functions in a program. Prototypes of so many functions cannot easily be extracted from large application binaries. The need for source code to obtain function prototypes renders it less practical for binary-only settings. The overall practicality of the solution is also limited by mctoll's limitations.

Dynamic Race Detection. Another possible approach is to employ a probabilistic data race detection tool like TSan [61] to detect racy memory accesses and to transform them into atomic sequentially consistent accesses. Its major drawbacks, however, are that of safety and practicality. It requires test cases to cover all the relevant code paths to work, which in turn requires a significant amount of user input/guidance. Moreover, the safety of the final result is limited by the tests that were actually run.

AtoMig: Practical Pattern-Based Porting. As we have seen, all these approaches either do not scale to larger software or incur a significant performance penalty, which opens up the problem space of this work. It is of high interest to find a solution, which helps in porting software designed for TSO to a WMM architecture. Specifically, the solution needs to be a) *scalable* to millions lines of code; b) *fix typical concurrency bugs* occurring on WMM architectures, c) be *automatically* applicable without manual test case curation, dynamic execution of the target software or change of the application source code and d) have *better performance* than the naïve conversion and state-of-the-art software solutions.

3 DESIGN AND IMPLEMENTATION OF ATOMIG

Our approach is to detect any memory accesses that are used for synchronization *statically* and to convert them to C/C++ sequentially consistent atomics to be turned into Arm's release/acquire atomic accesses that contain implicit barriers and avoid WMM behaviors. As previous work in this domain indicates that implicit barriers issued by atomic memory accesses are faster than explicit barriers [48], we use implicit barriers as often as possible and only fall back to explicit barriers at necessary places.

3.1 Overall Structure of AtoMig

We implement AtoMig as a set of LLVM link-time compiler passes, and integrate in the LLVM compilation/build chain, as depicted in Figure 2.

The application to be ported is initially compiled using its standard build system. We configure the build system via environment variables to use LLVM tools like clang, llvm-ar, llvm-ranlib and so on. The initial compilation is done exactly the way these applications are meant to be built, except that we use no optimizations "`-O0`" for the compilation to LLVM IR and pass some additional options to these tools in order to extract the LLVM IR from the linker to get a complete module for each build target. This initial run of the build system is as fast as the normal build of the application, if not faster, since no optimizations are performed.

When this initial compilation phase is finished, we run our analyses to detect synchronization accesses at the level of LLVM IR. We perform three major analysis passes: (1) *explicit annotation analysis*, (2) *implicit synchronization pattern detection*, and (3) *alias exploration*. AtoMig uses the result of these steps to mark the IR and transform the code to turn all detected racy memory accesses into atomic accesses. Once the transformation is complete, we apply any outstanding optimizations (e.g., `-O2` or `-O3`) and perform a final linking step to produce the ported binary. We describe each of these passes below.

3.2 Analyzing Explicit Annotations

As a first step, we look for any existing annotations at the application's source code, which hint at the use of shared memory to synchronize between threads. There are three such types of annotations we cover: (1) uses of C/C++11 `atomic` accesses, (2) uses of C/C++ `volatile` qualifier, (3) uses of inline assembly code.

The first kind is the easiest to deal with. Any atomic operations already found in the program invariably indicate the presence of concurrent accesses on them. Since, however, on TSO, most of the attached memory orders on accesses — relaxed, consume, acquire, release, and even SC for anything other than a write — are indistinguishable, it is frequent for code to use insufficiently strong memory orders. To ensure correctness under WMM, we therefore turn all of these memory orders into SC.

The second kind is arguably the most common means of indicating shared-memory accesses in legacy code. Developers typically annotate shared variables with the volatile qualifier in order to suppress compiler optimizations on them and preserve intended concurrent behavior. By default, standard compiler optimizations (e.g., CSE) assume the program is sequential, and can easily break concurrent code, whose variables are not annotated as volatile or atomic. Unlike atomics, the volatile qualifier, however, has no influence on how the hardware treats those accesses. Unless we make them atomic accesses, say, using the compiler built-ins or using atomic instructions, the hardware does not provide any extra consistency guarantees for concurrent accesses on them. We therefore convert all accesses to volatile variables into sequentially consistent atomic accesses.

Variables are generally marked as volatile because they are subject to concurrent accesses either (1) from the program itself or (2) from its environment, e.g., from peripheral hardware devices, in the case of interrupted executions like signal handlers and device drivers. Since we are only interested in accesses of the first kind, to avoid additional overheads, we can exclude the latter kind from our transformation through blacklisting. Throughout all experiments that we performed, blacklisting of volatile variables was never necessary.

Finally, developers often implement synchronization barriers with architecture-specific assembly instructions, for performance reasons or because there was no other way to introduce barriers when the code was first written. Porting such inline assembly code segments to a different architecture is particularly challenging because one needs a mapping to the new architecture not only of the compiler's IR but also of all the x86 instructions. Further, it remains out of reach of our static analysis at the IR level. To address these challenges, we develop a compiler frontend pass that analyzes all uses of x86 inline assembly implementing synchronization patterns in the source code and replaces them with their compiler builtin counterparts. Compiler built-ins are amenable to LLVM-based static analysis and for porting, we can simply let the compiler generate target specific instructions.

If all racy accesses are already correctly annotated, applying these steps will result in a correctly ported application without WMM consistency bugs. Sadly, this is rarely the case in large applications. We therefore need another step to detect synchronization accesses that the developers 'forgot' to annotate.

3.3 Detecting Implicit Synchronization Patterns

To find unannotated potentially racy accesses, AtoMig detects synchronization code patterns and transforms them. In the figures, we mark AtoMig transformations in orange, with -> SC for adding an implicit memory barrier and —— FENCE SC —— for an explicit barrier.

```
int flag = WAIT, turns = 7;
void spinloop_examples() {
  int l_flag, l_turns = 7;

  // Spinloop 1
  while(flag != DONE) ;      // non-local dep.

  // Spinloop 2
  do {
    l_flag = DONE;           // constant store
  } while (l_flag != flag);  // non-local dep.

  // Spinloop 3
  do {
    l_flag = flag & F_MASK;  // non-local dep.
  } while (l_flag != READY); // in-loop dep.

  // Non-spinloop: has local exit condition
  for (int i=0; i < 100; i++)
    if (flag == DONE) break;

  // Non-spinloop: exit depends on local store
  for (int i=0; i < turns; i++) ;
}
```

Figure 3: Examples of spinloops and non-spinloops

Spinloops. The most basic pattern is that of a spinloop, where a thread waits in a loop for some condition to hold, which is not updated by the loop itself. So, the only way to exit a spinloop is if some other thread changes the program state and invalidates the spinloop's wait condition. That is, a spinloop is a loop where all its exit conditions depend on operations external to the current thread.

Let us formalize these notions a bit. A *loop* is identified by its loop header, a node in a program's control-flow graph (CFG) that has an incoming backedge, and contains all nodes that are dominated by the loop header and which have a path back to the loop header. A *loop exit condition* is any condition on a branch that exits the loop. A memory access is *non-local* in a function if it may also be accessed from outside that function; e.g., a global variable, a function argument passed by reference, or a stack variable whose address is taken and escapes the function scope. A CFG node has a *non-local dependency* if it depends on some non-local access either directly or indirectly (i.e., by using a variable whose definition has a non-local dependency). A loop is a *spinloop* if (1) all its exit conditions have *non-local dependencies*, and (2) all the stores in the loop without *non-local dependencies* do not influence the loop exit conditions.

Figure 3 presents some examples and non-examples of spinloops. The first loop is clearly a spinloop because its exit condition depends solely on the value of a global integer, flag and there are no operations within the loop that affect its exit condition. Similarly, the exit condition of second loop has a non-local dependency on

```
int locked = 0;

void lock() {
    while(!cmpxchg(locked -> SC,
                   0, 1));
}

void unlock() {
    locked = 0; -> SC
}
```

Figure 4: Test-and-set mutual exclusion lock.

```
                int flag = WAIT;
                int msg;

// reader         || // writer
int data;         ||
while(            || msg = get_msg();
  flag -> SC      || flag = DONE; -> SC
  != DONE);       ||
data = msg;       ||
```

Figure 5: Message passing using a spinloop.

flag. The store inside the loop cannot influence the exit condition, because although the condition depends on it, it always writes the same constant value to l_flag. The third loop's exit condition has an indirect non-local dependency through l_flag, and is therefore also a spinloop.

Next, consider two non-spinloop examples. The first for loop contains two exit conditions, one with the non-local dependency on flag but also one without any non-local dependencies. The second for loop's exit condition has a non-local dependency on the global variable turns, but is also affected by the i++ store inside the loop, which has only local dependencies. Therefore, neither of these loops are spinloops. Both these loops are guaranteed to eventually terminate regardless of what other threads do. In the first case, the loop will run for at most 100 iterations. While in the second case another thread can delay the termination of the for loop. Eventually the thread increasing i will catch up and exit the loop.

For each spinloop, AtoMig performs a static *instruction-influence analysis* to identify all non-local memory accesses that influence the loop exit conditions and marks all those non-local memory accesses (as opposed to accesses on their copies on the local stack) as *spin controls*, which in a subsequent pass will be transformed into sequentially consistent atomic operations.

Once Atomic, Always Atomic. Transforming only the accesses detected within spinloops to SC-atomics is not sufficient. We must also transform all the accesses to *spin control*-variables to atomics, whether they appear inside a loop or not. We illustrate this point with two examples.

Figure 4 presents a trivial test-and-set lock. To acquire the lock, a thread repeatedly performs an atomic compare-exchange in a spinloop. The compare-exchange updates the locked variable to 1 and returns the old value, i.e., the loop exits when the lock was previously free. To release the lock, a thread unlocks by simply setting locked to 0. This lock implementation works on an x86 machine because TSO guarantees that all operations of a critical section remain ordered between the compare-exchange and the store of the lock release. This guarantee, however, is not provided by WMM CPUs. If we only make the lock acquisition SC-atomic, the operations of a critical section may be executed after the release of the lock, which is clearly wrong. For a correct operation, we therefore also need to make the locked=0 store atomic.

As a second example, consider "message passing" pattern in Figure 5, which appears very frequently in concurrent code. For example, the MCS lock [55] uses such a loop to wait for other threads.

Since the order of execution on stores is preserved on x86, the update to DONE on flag always executes after the writer writes to message. So, if the reader reads DONE in flag and exits the loop, the value of message read will be the correct value, as expected. WMM CPUs, however, allow the reordering not only of the loads within the while loop and the load from message on the reader side and, but also of the two stores on the writer side. To prevent undesirable reordering of loads and stores, we can again make the flag variable *atomic* (sequentially consistent) *throughout the whole program*. On the reader side, the atomic loads of flag prevents the subsequent loads to be executed before the loop and on the writer side, the atomic flag=DONE write prevents the write to message from executing after it.

Optimistic Accesses. The transformations described so far work for plain spinloops, but are insufficient for code with *optimistic concurrency control* [34, 35, 45], such as sequence locks [53]. Consider the code in Figure 6. The writer increases a sequence counter flag, writes to message, and increases the sequence counter again, while the reader continuously reads the message and the sequence counter in a spinloop and exits whenever it manages to read twice the sequence count holding the same even value. The scheme helps in improving performance by reducing the points of synchronization among the concurrent threads. Since the ordering between stores and between loads is preserved on x86-TSO, reading twice the same even value means that the writer did not update the message variable concurrently, and so the read's in-between load of message returned the latest value.

Under WMM, however, neither of these orderings are preserved, and so this optimistic scheme is broken. Moreover, even if we make all accesses of flag SC-atomic (e.g., by running the spinloop-detection phase as described earlier), the optimistic scheme remains incorrect because the read of message can be reordered after the subsequent SC-atomic load of the flag variable.

Let us now see how such loops with optimistic concurrency control — *optimistic loops*, for short — can be detected. The general pattern for such loops is to optimistically read certain other shared variables besides their spin controls, and for those optimistic reads

```
                 volatile int flag = 0;
                 int msg;

// reader                    ║ // writer
int i, data;                 ║
do {                         ║
 ——— FENCE SC ———            ║
  i = flag -> SC ;           ║ flag++; -> SC
  data = msg;                ║  ——— FENCE SC ———
 ——— FENCE SC ———            ║ msg = get_msg();
} while( i%2 != 0 ||         ║ flag++; -> SC
  i != flag -> SC );         ║  ——— FENCE SC ———
                             ║
process(data);               ║
```

Figure 6: Sequence count

to be used after the loop. In the case of sequence counting, these optimistic reads, moreover, appear between repetitive reads to the spin controls. More formally, a spinloop is called an *optimistic loop* if it contains a read of a non-local variable different from all the spin controls that is used by some operation outside the loop. We mark all the spin controls of an optimistic loop as *optimistic controls*.

Optimistic accesses to non-local memory sources must execute within the spinloop it belongs to. At the exit conditions of an optimistic loop, we can find loads to the optimistic controls of the loop. Since an optimistic loop is a spinloop, the optimistic controls are already slated to be sequentially consistent atomic accesses. We also insert an explicit memory barrier before their loads to force all the uncontrolled reads to non-local memory within the loop to execute before exiting the loop. At the writer side, the stores to the uncontrolled non-local source must execute after the store to *optimistic control* (the first `flag++`) and there should not be any reordering with any instructions that follow the write to optimistic control. Therefore, we add a sequentially consistent explicit barrier after the store to the optimistic control. Ensuring that the optimistic reads stay within the spinloop and writes to the optimistically accessed memory locations execute after the update to the optimistic controls, protects the programmer intended order of such concurrent executions.

In summary, AtoMig detects spinloops and adds spin control marks to their non-local exit dependencies. It then checks all spinloops for optimistic accesses and if so, marks those spin controls as optimistic controls. It turns all spin control and optimistic control memory accesses into atomic SC accesses. For the latter, it also inserts an explicit barrier before each optimistic control read within optimistic loops and after all optimistic control stores.

3.4 Alias Exploration

As already said, AtoMig's transformations apply not only to the spin controls and optimistic controls within spinloops, but throughout the entire program. To implement this transformation, we perform a complete link-time pass at the program's module scope to detect all atomic accesses. For each detected atomic access, we statically look for other instances of accesses to these identified memory locations and mark them as their *sticky buddies*. Finding sticky buddies of accesses to global variables is straightforward but pointer based accesses to global and heap memory locations is not. For the latter, we apply a type based detection scheme. These accesses are to the following kinds of memory locations: `struct` typed entities or member fields and arrays. LLVM represents such accesses using the `getelementptr` instruction. This instruction tells us the type of the access along with its offset information. We find their sticky buddies by finding all other `getelementptr` instructions in the program that have the same type and offsets. Alternatively, we can apply alias analysis to detect sticky buddies of pointer-based accesses in the program. While both the approaches can report false positives, alias analysis needs inter-procedural analysis and can easily result in memory exhaustion [16] for large code bases. Therefore, we choose a type-based detection scheme in the interest of scalability. If synchronization involves generic data types, our type-based alias analysis can result in larger sets of sticky-buddies. However, typically synchronization points involve simpler data types designed for concurrent updates (e.g., to update flags, values or pointers). Finally, we turn all *sticky buddies* into SC-atomic accesses. Similarly, we also find *sticky buddies* of all detected spin controls and optimistic controls and perform the same respective transformations for their *sticky buddies*. In other words, sticky buddies of spin controls turn into SC-atomic operations and the sticky buddies of optimistic controls additionally get explicit barriers depending on where they are in the code base.

While existing annotation and implicit synchronization pattern detection marks the start of our static exploration of synchronization code in a target application, finding sticky buddies expands our purview to the rest of the code base, allowing us to automatically prevent undesirable reorderings of instructions on a WMM architecture.

3.5 Static Analysis Challenges

We need AtoMig's compiler-based static analyses to be efficient to scale for large code bases. Both the time taken for instrumenting an application and the overall memory usage must remain reasonable for practical integration with existing build infrastructure. We discuss below how we address the challenges in the design of AtoMig.

Instruction Influence Analysis. All schemes that AtoMig employs to detect susceptible patterns, heavily rely on detecting dependencies between memory locations, their scopes and their influence on dependencies among instructions in the target program. For example, for spinloop detection, we must chase all the loads of loop compare instructions, and check scopes of the accessed memory locations if they are function-local or not. Some stores may affect the memory locations that the `load` instructions load from, expanding the set of dependent instructions and hence the set of memory sources that have influence on the compare instructions. Trivially following the dependencies across the entire application and frequently doing so, will be highly expensive. Owing to our choice of spinloops as entry point to synchronization detection and all the susceptible patterns from there on, we only need *fine-grained* memory dependency analysis, i.e., scoped within a few specified basic blocks, a loop or at most within a function. We exploit this to implement an intra-procedural static data flow analysis across loads and stores to memory locations (technique akin to memory dependence analysis [6] albeit finer-grained) for extracting scope and instruction-influence information within the specified program regions. AtoMig applies such fine-grained scoping wherever appropriate. For example, to find influence of intra-loop stores on the loop compare instructions, it is sufficient to scope the analysis to just within the loop. Moreover, AtoMig's static transformations do not invalidate these existing dependencies that we detect. Therefore, we also cache the results to speed up any further queries of memory sources, their scope information and instruction influence relations.

Type-Based Pointer Alias Analysis. Inherent limitations of static analyses (including pointer alias analysis) can cause memory exhaustion. AtoMig avoids them by applying a type-based pointer alias analysis, to find all memory accesses to a specific memory location to implement *sticky atomics*. We use the type, base and offsets of `getelementptr` instructions in the LLVM intermediate representation of the program and create a module-wide map of all such memory accesses. Finding similar memory accesses thus reduces to a constant time access to this map. Since our transformations do not change any relations between memory accesses, we only have to populate this map once during initialization. Further, a memory access instruction, once *stickied*, remains so forever. So, we mark them to prevent processing them repetitively.

Loops Spanning Multiple Functions. Loops that span across several functions pose a harder challenge for static analysis. To avoid the inherent precision limitations of inter-procedural analysis, we inline functions where possible beforehand. This offers a better practical trade-off between performance, scalability and correctness.

False Negatives and Positives. False negatives are critical for the application correctness. In AtoMig they can mostly arise when a synchronization point is using plain reads and writes and is not detected by any other employed heuristic.

There are not many common lock-free data structures or synchronization patterns that can be constructed using only read/write registers. It is shown [36] that for wait-free algorithms, a read-modify-write register is necessary to construct non-trivial synchronization algorithms. While this result limits the construction of wait-free algorithms, typical implementations of lock-free algorithms are using read-modify-write operations to a large extent. For example in CK [15] more than 80% of the algorithms use read-modify-write operations. Any use of an atomic read-modify-write operation will be detected and propagated. False negatives also likely decrease with bigger code bases due to alias exploration becoming more effective since it gets easier to find at least one atomic access to each shared memory location. Further, there are higher chances of our pattern detection to catch at least one of the loops that depend on a specific shared variable. Alias exploration can transform the rest. However, false negatives of alias exploration due to dynamically changing addresses hinder its effectiveness.

Our choice of spinloop defition was motivated by the fact that the overall goal is to reach a practical trade-off between performance and correctness. We are aware that other definitions of a spinloop exist in literature, however, they are more restrictive compared to our definition, therefore we assume that using these definitions would result in fewer detected synchronization points and thus a less correct ported application.

False positives can only affect the performance of the application, not its correctness. Typically, threads of a well-designed concurrent application spend most of their CPU cycles performing concurrent work, rather than synchronizing with other threads. AtoMig mainly introduces overhead in parts of the code that perform synchronization, which in turn are shorter and (hopefully) constitute a small fraction of the total execution time. Even though incorporating a precise inter-procedural data flow analysis would reduce the number of false positives, the end-to-end performance is unlikely to be affected significantly. Moreover, such a precise data-flow analysis could limit the scalability of our tool to large code bases.

Table 2: Verification results on ck and lf-hash

	Original	Expl.	Spin	AtoMig
ck_ring	✗	✓	✓	✓
ck_spinlock_cas	✗	✓	✓	✓
ck_spinlock_mcs	✗	✗	✓	✓
ck_sequence	✗	✗	✗	✓
lf-hash	✗	✗	✗	✓

4 EVALUATION

In this section, we evaluate AtoMig for correctness, scalability, and performance.

4.1 Correctness

To evaluate the correctness of AtoMig transformations, we employ GenMC [42], a state-of-the-art stateless model checker that exposes weak memory model concurrency issues in the target applications. Since model checking does not scale to large code bases, we use smaller benchmarks for our evaluation and make necessary adaptations to run GenMC on them (e.g., creating client programs exercising the module). Specifically, we choose as benchmarks:

(1) a few concurrent benchmarks from Concurrency Kit [15], a widely-used library implementing efficient concurrent data structures, which supports different memory models for appropriate placement of barriers;
(2) the lock-free hash-table [63] (lf-hash), a key component in MariaDB, whose correctness is paramount to everything that is built around it.

We report the results of our evaluation in Table 2. We run GenMC on *original* (x86-TSO) and AtoMig-instrumented variants of these benchmarks, and check whether an assertion violation is reported. We also run GenMC on versions of the benchmarks generated by AtoMig by switching off certain parts of its detection of synchronization patterns: *Expl.* uses the explicit annotations, *Spin* also uses spinloop detection, while AtoMig also uses optimistic loop detection. As it can be seen, explicit annotations are rarely sufficient for

Table 3: AtoMig statistics for large applications showing synchronization pattern detection, time taken for original build & AtoMig and explicit(B_{Expl}) & implicit(B_{Impl}) barriers present (SLOC: source lines of code).

Applications	# SLOC	# Spinloops	# Optiloops	Build Time		Original		AtoMig		Naïve
				Original	AtoMig	# B_{Expl}	# B_{Imp}	# B_{Expl}	# B_{Impl}	# B_{Impl}
MariaDB	3,124,265	12,880	1,970	20m 51s	40m 21s	0	968	12,361	66,347	366,774
PostgreSQL	880,400	1,750	544	4m 59s	10m 40s	104	340	3,455	42,744	243,790
LevelDB	82,725	458	263	1m 17s	3m 21s	0	390	2,798	11,128	65,042
Memcached	28,957	75	20	17s	30s	2	0	231	1,564	11,515
SQLite	263,125	1,057	254	4m 1s	11m 54s	1	28	4,016	44,860	122,611

Table 4: AtoMig statistics on Memcached for number of dynamically executed barriers.

Memcached	Original	AtoMig
non-atomic loads	376,833,638	357,562,200
non-atomic stores	127,279,768	117,914,988
atomic loads	0	19,975,815
atomic stores	0	5,458,192

```
l_find() {
  do {
     —— FENCE SC ——
    state = node->state -> SC ;
    key = node->key;
     —— FENCE SC ——
  } while (state != node->state -> SC &&
           state == INVALID);
  assert(key != NULL);
}

l_delete() {
  if (cmpxchg(node->state, VALID, INVALID)) {
     —— FENCE SC ——
    node->key = NULL;
  }
}
```

Figure 7: lf-hash WMM bug in MariaDB

correct porting, spinloop detection works correctly for a few more cases, while AtoMig handles all the benchmarks correctly.

MariaDB Hash-Table Bug. We remark that while evaluating the AtoMig-port of the lock-free hash-table benchmark, we also found a serious WMM concurrency bug in MariaDB's implementation, which was manually ported from the x86 to WMMs.

Figure 7 abstracts the bug. One thread operates on a node within `l_find`, while another thread operates on the same node within `l_delete`. Two problematic reorderings are possible on a WMM architecture like Armv8. Within `l_find`, the load of `node->key` can be reordered with the subsequent load of `node->state`, possibly using an old value of `state` with an updated value of `key`. On Armv8, a `cmpxchg` consists of two memory accesses, a load and store with acquire/release semantics. As the store release can be reordered with subsequent memory operations, the relaxed store to `node->key` within `l_delete` can be visible before the `cmpxchg` store.

To fix this algorithm, all operations on `node->state` should follow acquire/release semantics and thus the loads in `l_find` are made atomic SC by AtoMig. In addition, all operations on `node->key` should have acquire/release semantics. As our framework detects such optimistic load patterns and marks `node->state` as an optimistic control, it introduces explicit barriers to protect against reorderings of optimistic loads and belonging stores. These explicit barriers also protect against reorderings of memory accesses that could not be detected statically, i.e., dynamic addresses calculated at runtime through pointer arithmetic.

Upon confirming the bug using GenMC and reporting the bug, our proposed fix has been accepted and merged into the current MariaDB code base [4].

4.2 Scalability

For evaluating scalability, we apply AtoMig to a collection of popular large-scale applications that are widely deployed as critical components of larger infrastructure viz., MariaDB [5], PostgreSQL [10], LevelDB [8], Memcached [9] and SQLite [11].

Table 3 reports the number of patterns AtoMig detected and transformed in each of the large-scale applications, as well as the AtoMig porting time. A first observation from this table is that these large applications contain a very large number of synchronization patterns, which goes beyond the limits of what can be analyzed effectively by human experts. Secondly, applying AtoMig to a project increases the build time by a factor between 2 and 3 compared to the original build time. For example, building the complete, default MariaDB project with parallel compilation, takes around 20 minutes on our test infrastructure. Applying AtoMig to it took in total about 40 minutes. Maximum RAM usage during AtoMig builds still comes from compilation, optimization and linking steps, so AtoMig does not increase maximum memory usage.

We also report the number of implicit barriers added by AtoMig. These are explained by the number of spinloops found in the code base in combination with alias exploration to make all occurrences of spin control accesses atomic. Interestingly, even such a high number of implicit barriers does not cause a large performance impact, likely due to the higher performance of implicit barriers compared to explicit barriers [48]. Similarly, the number of explicit barriers relates to the detected optimistic loops and alias exploration thereon. Much less explicit barriers are added compared to implicit barriers, which follows nicely from our design decision. The table also reports the number of explicit and implicit barriers present in the original code base before application of AtoMig.

For comparison, we have also measured the numbers of dynamically executed barriers during the Memcached benchmark, and presented them in Table 4. The measurement was performed for the original and AtoMig version of Memcached.

4.3 Performance

We next evaluate the impact on performance by the barriers introduced by AtoMig to restore correctness under WMMs. We compare our tool with the only other similarly scalable strategy, namely the *naïve* strategy of making all global memory accesses SC-atomic. Our performance evaluation experiments compare aarch64 binaries produced by these two approaches. We report the slowdown with respect to their *original* aarch64 binaries compiled from sources, applying standard optimizations. We measure their performance on a 96-core server with 2 x HiSilicon Kunpeng 920 (Armv8.2) 2.6 GHz CPUs, 128 GiB RAM, running OpenEuler 20.03 [3] Linux distribution.

As benchmarks for the comparison, we choose:

- the large-scale applications used for demonstrating scalability of our approach;
- the smaller concurrent data structures used in the correctness evaluation; and
- CLHT [24], a concurrent hash-table library developed solely for x86, in order to demonstrate the possibility of end-to-end porting to aarch64 with AtoMig.

To evaluate performance on MariaDB, we use the tests from its `mtr` framework to record their run times. For PostgreSQL, we use its `pgbench` tool and for LevelDB, we use its `db_bench` tool included in its source distribution. For Memcached, we use the Memtier-benchmark [2]. For CK, we use micro-benchmarks that perform parallel operations on the same set of data structures used for correctness evaluation and compare performance of AtoMig with expert porting, by comparing AtoMig instrumented TSO aarch64 builds with their weak memory aarch64 counterparts. To evaluate performance of the extracted lf-hash data structure, we use a client that performs parallel searches, insertions and deletions upon the hash map. Finally, for CLHT, we use the lock-based and lock-free versions of its `test_mem` benchmark. Since CLHT does not have a version of the code that is ported for Arm, as a baseline we simply recompile the source code to aarch64 without making any adjustments whatsoever for the weak memory model, which is bound to exhibit undesirable WMM effects.

Table 5 summarizes our performance results. As we can see, AtoMig provides very low performance overhead on the large benchmarks (0%–4%, 1.8% on average), while the naïve strategy has substantially higher overhead, except on Memcached. The Concurrency Kit benchmarks exhibit some unintuitive behavior, where the AtoMig-ported binary performs repeatably much better than the native expert-ported WMM implementation. This happens because AtoMig adds implicit barriers to the TSO version, whereas the expert-ported version uses explicit barriers, which are substantially slower. Observing that for these CK benchmarks, the AtoMig-generated version has been formally verified for its correctness, one might reasonably argue porting should be left to machines rather than to humans. Our evaluation certainly backs that claim. Also note that the naïve strategy performs especially poorly for these CK benchmarks. Finally, the larger performance overhead of AtoMig on CLHT can be attributed to the fact that CLHT benchmark does not have an appropriate WMM version, so we use an incorrect version as a baseline.

Table 5: Performance impact comparing *Naïve* and *AtoMig* variants, normalized with the *original* versions.
$^+$ The *originals* for CLHT have no WMM corrections.

	Naïve	AtoMig
MariaDB [5]	1.27	1.01
PostgreSQL [10]	1.35	1.04
LevelDB [8]	1.66	1.01
Memcached [9]	1.01	1.00
SQLite [11]	2.49	1.03
ck_ring	4.43	0.85
ck_sequence	5.35	0.91
ck_spinlock_cas	3.75	0.63
ck_spinlock_mcs	5.29	0.64
lf-hash	3.05	1.01
clht_lb$^+$	1.89	1.10
clht_lf$^+$	2.01	1.40

Comparison with Lasagne. We finally compare the performance of AtoMig against Lasagne [60]. Due to difficulties in running Lasagne on our benchmarks, we use the Phoenix 2.0 benchmark suite, which was used in Lasagne's artifact and consists of map-reduce programs that are widely used to benchmark parallel executions.

Table 6 compares the normalized performance impact of AtoMig with the naïve approach and Lasagne [60]. The naïve scheme of protecting all memory accesses with implicit barriers has a significant performance impact. In these parallel benchmarks, however, almost all of these implicit barriers are unnecessary. The threads in these programs generally only synchronize using `pthread` based barriers (i.e., not based on shared memory accesses) in between performing trivially parallel tasks, which makes AtoMig's pattern-based strategy exhibit an almost negligible performance impact. Quite remarkably, however, Lasagne performs substantially worse than the naïve solution (16% slower on average). The reason for this is that Lasagne performs binary lifting to insert explicit barriers between memory operations and then applies a sequence of formally verified barrier optimizations to remove any provably redundant barriers. For this benchmark suite, however, these barrier elimination optimizations are not sufficient for compensating for the much higher cost incurred by using explicit barriers instead of implicit ones, as used by the naïve approach and AtoMig.

In summary, the results clearly shows that our approach is scalable to large code bases like MariaDB with several million lines of code as shown in Table 3 and induces low runtime performance impact, much lower than an naïve approach, while still fixing real world concurrency bugs, as depicted in the correctness evaluation.

Table 6: Performance results for the Phoenix benchmark. Results are normalized to the original performance of each benchmark. They are given as a factor of slowdown.

	Naïve	Lasagne	AtoMig
histogram	2.80	2.51	1.00
kmeans	1.07	1.60	1.03
linear_regression	1.02	1.90	1.00
matrix_multiply	1.01	1.49	1.01
string_match	1.70	1.35	1.01
geometric mean	1.39	1.73	1.01

5 RELATED WORK

Dynamic Binary Translation. There are several dynamic binary translation techniques that support cross-architecture execution of programs [23, 28, 38, 65, 69]. However, they focus on efficient ISA-level translations and often remain oblivious to memory model differences between the two architectures. Apple's Rosetta 2 [7, 56] performs dynamic binary translation to run x86 programs on the current generation of Apple's Arm-based CPUs. For the difference in memory models, it relies rather on the CPU's ability to switch to TSO mode. While proprietary software and hardware dependencies limit its wider applicability, AtoMig is completely a software solution that is suitable for any WMM-based platform. On the other hand, ArMOR [52] performs dynamic memory-model translations. It provides architecture-independent expression of memory-model differences and applies it to dynamically inject barriers when running cross-memory model programs in a heterogeneous architecture setting. Its significantly higher performance overhead is due to rather conservative insertion of many barriers.

Static Approaches. Prior research on preserving SC in compiler optimizations [54] and also in the hardware [64], eliminate WMM problems by design. In contrast, we turn detected concurrent code to SC for running correctly on WMM hardware. There are other approaches that use static analysis to apply checks based on axiomatic rules [16] or robustness checking [21] to find WMM bugs and perform barrier insertion. Inherent limitations of static analysis and imprecise alias analysis, limits scalability of these static approaches. Lasagne [60] lifts an x86 binary to LLVM IR and inserts barriers for correct execution on Arm. Limitations of binary lifting hinders its applicability on large applications and the limited opportunities for barrier removal in a binary-only setting, adds higher performance overhead than AtoMig.

Model Checking. VSync [57] performs verified and optimized placement of memory ordering enforcements for synchronization primitives for WMM, by employing model checking on barrier mutations. However, model checking does not scale for large programs. While stateful model checkers [13, 22, 37, 40, 46, 58, 72] quickly run into state explosion problems, even stateless model checkers [12, 14, 17, 25, 41–44] cannot scale for code sizes larger than a few thousand lines.

6 DISCUSSION AND CONCLUSIONS

In this paper, we have presented AtoMig, which is the first *practical* software-based approach to porting large applications from x86-TSO to WMM. Motivated by the needs of the industry, our approach sacrifices formal correctness guarantees in order to scale to millions of lines of code and achieve very low performance overheads. We argue that this trade-off is worth taking because existing approaches that provide correctness guarantees do not scale [16, 57] and/or generate code with a high performance overhead [60]. Moreover, to the point that it can be formally evaluated, AtoMig generates correct code on practical use cases (cf. §4.1).

We note that there is a subtle balance between detecting too many synchronization patterns, which will in turn introduce too many barriers hampering performance, and too few, which can produce incorrect code. Our focus on loop-induced synchronization is motivated by the fact that (1) loop patterns are much more common, (2) are easier to detect with a low false positivity rate than patterns on straight-line code, (3) more frequently experience WMM issues. The latter point follows both from the relative infrequency of WMM effects [18] and from our own experience with WMM-related bug reports. As such, our implementation does not currently find other synchronization points that cannot be traced back to a variable used in a spinloop. Possible examples of such synchronization points could be shared memory accesses mixed with timing-based polling or asynchronous methods like signals or system calls akin to sleep semantics. Locating code segments around specific system calls or external library functions that offer wait semantics can help in their detection, and can easily be integrated in our approach. Another idea worth exploring is to use the placement of compiler barriers (which are turned into NOPs in the generated assembly code) as additional entry points for detecting synchronization points.

Note, moreover, that our spinloop definition is *conservative* in that it covers only loops that require external help to terminate. Yet, there are often synchronizing loops that choose to terminate after a fixed number of iterations for various efficiency reasons, resembling the non-spinloop examples in Figure 3. The problem is that designating such loops as spinloops generates a huge number of false positives, since all loops with an exit condition influenced by a global variable (e.g., sequential search algorithms) would be treated as spinloops. Moreover, in practice, directly detecting such synchronization loops is completely unnecessary in the common case when their variables are also used in normal spinloops. In such cases, alias exploration covers them.

One final limitation of our approach (along with all other existing software-based approaches) is that we require the application source code to be available and, moreover, the application should be capable of being built with LLVM/clang. This is generally the case for open-source C/C++ projects, but not necessarily so for large-scale industrial code bases. To handle such scenarios, in the future, we plan to integrate our approach with a static binary translation tool like `mctoll` [71].

AtoMig serves as a framework that we plan to extend with further patterns to detect other synchronization points. We want to experiment with other efficient alias analysis algorithms, like those included in the LLVM distribution or SVF [66].

REFERENCES

[1] spin_unlock optimization(i386). https://marc.info/?l=linux-kernel&m=94318921016232&w=2, 1999. Accessed: 2022-07-06.

[2] Throughput benchmarking tool for redis & memcached. https://redis.com/blog/memtier$_b$enchmark-a-high-throughput-benchmarking-tool-for-redis-memcached/, 2013. Accessed: 2022-07-06.

[3] openeuler. https://openeuler.org, 2020. Accessed: 2022-07-06.

[4] Lock-free hash table bug fix in MariaDB. https://jira.mariadb.org/browse/MDEV-27088, 2021. Accessed: 2022-07-06.

[5] Mariadb. https://mariadb.com, 2021. Accessed: 2022-07-06.

[6] -memdep: Memory dependence analysis. https://llvm.org/docs/Passes.html#memdep-memory-dependence-analysis, 2022. Accessed: 2022-07-05.

[7] About the rosetta translation environment. https://developer.apple.com/documentation/apple-silicon/about-the-rosetta-translation-environment, 2022. Accessed: 2022-07-06.

[8] Google leveldb. https://github.com/google/leveldb, 2022. Accessed: 2022-07-06.

[9] Memcached. https://memcached.org, 2022. Accessed: 2022-07-06.

[10] Postgresql. https://www.postgresql.org, 2022. Accessed: 2022-07-06.

[11] Sqlite. https://www.sqlite.org, 2022. Accessed: 2022-07-06.

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

[13] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. 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*, page 56–74, Berlin, Heidelberg, 2017. Springer-Verlag.

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

[15] Samy Al Bahra. Concurrency kit. https://github.com/concurrencykit/ck, 2015. Accessed: 2022-07-06.

[16] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 508–524, Cham, 2014. Springer International Publishing.

[17] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *International Conference on Computer Aided Verification*, pages 141–157. Springer, 2013.

[18] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: Running tests against hardware. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 41–44, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[19] Amazon Web Services. AWS Graviton Processor – Enabling the best price performance in Amazon EC2. https://aws.amazon.com/ec2/graviton, 2020. Accessed: 2022-07-06.

[20] Arm. Introducing the Arm architecture, ARM062-948681440-3277. https://developer.arm.com/architectures/learn-the-architecture/introducing-the-arm-architecture/about-the-arm-architecture, 2019. Accessed: 2022-07-06.

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

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

[23] Emilio G. Cota, Paolo Bonzini, Alex Bennée, and Luca P. Carloni. Cross-ISA machine emulation for multicores. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 210–220, 2017.

[24] Tudor Alexandru David, Rachid Guerraoui, Tong Che, and Vasileios Trigonakis. Designing ascy-compliant concurrent search data structures. page 23, 2014.

[25] Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. BMC with memory models as modules. In *FMCAD 2018*, pages 1–9. IEEE, 2018.

[26] Will Deacon. 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, Feb 13, 2018. Accessed: 2022-07-06.

[27] Abdullah Diaa. Is Apple Silicon ready? https://isapplesiliconready.com, 2022. Accessed: 2022-07-06.

[28] Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, and Yeh-Ching Chung. PQEMU: A parallel system emulator based on QEMU. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, pages 276–283, 2011.

[29] Linux Foundation. Data Plane Development Kit (DPDK), 2015. Accessed: 2022-07-06.

[30] Geekbench. Geekbench 5 - Cross-Platform Benchmark. https://geekbench.com, June 2021. Accessed: 2022-07-06.

[31] Geekbench. Geekbench 5.4.1 Tryout for macOS AArch64. https://browser.geekbench.com/v5/cpu/8239789, June 2021. Accessed: 2022-07-06.

[32] Geekbench. Geekbench 5.4.1 Tryout for macOS x86 (64-bit). https://browser.geekbench.com/v5/cpu/8252865, June 2021. Accessed: 2022-07-06.

[33] Samuel Greengard. Will RISC-V revolutionize computing? *Commun. ACM*, 63(5):30–32, April 2020.

[34] Theo Härder. Observations on optimistic concurrency control schemes. *Information Systems*, 9(2):111–120, 1984.

[35] Maurice Herlihy. Optimistic concurrency control for abstract data types. In *Proceedings of the fifth annual ACM symposium on Principles of distributed computing*, pages 206–217, 1986.

[36] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, jan 1991.

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

[38] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. HQEMU: A multi-threaded and retargetable dynamic binary translator on multicores. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, page 104–113, New York, NY, USA, 2012. Association for Computing Machinery.

[39] Huawei. Huawei Unveils Industry's Highest-Performance ARM-based CPU. https://www.huawei.com/en/news/2019/1/huawei-unveils-highest-performance-arm-based-cpu, Jan 2019. Accessed: 2022-07-06.

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

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

[42] 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.

[43] Michalis Kokologiannakis and Konstantinos Sagonas. Stateless model checking of the linux kernel's hierarchical read-copy-update (tree rcu). In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, SPIN 2017, page 172–181, New York, NY, USA, 2017. Association for Computing Machinery.

[44] Michalis Kokologiannakis and Viktor Vafeiadis. HMC: Model checking for hardware memory models. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1157–1171, New York, NY, USA, 2020. Association for Computing Machinery.

[45] Hsiang-Tsung Kung and John T Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

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

[47] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers c-28*, 9:690–691, 1979.

[48] Nian Liu, Binyu Zang, and Haibo Chen. *No Barrier in the Road: A Comprehensive Study and Optimization of ARM Barriers*, page 348–361. Association for Computing Machinery, New York, NY, USA, 2020.

[49] Waiman Long. locking/qspinlock: Use _acquire/_release() versions of cmpxchg() & xchg(). https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=64d816cba06c, Nov 10, 2015. Accessed: 2022-07-06.

[50] Waiman Long and Peter Zijlstra. 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, 2015. Accessed: 2022-07-06.

[51] Waiman Long and Peter Zijlstra. 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, 2020. Accessed: 2022-07-06.

[52] Daniel Lustig, Caroline Trippel, Michael Pellauer, and Margaret Martonosi. Armor: Defending against memory consistency model mismatches in heterogeneous architectures. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 388–400, New York, NY, USA, 2015. Association for Computing Machinery.

[53] LWN. Driver porting: mutual exclusion with seqlocks. https://lwn.net/Articles/22818, Feb 2003. Accessed: 2022-07-06.

[54] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. A case for an sc-preserving compiler. *ACM SIGPLAN Notices*, 46(6):199–210, 2011.

[55] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991.

[56] Koh Nakagawa. Reverse-engineering rosetta 2 part1: Analyzing aot files and the rosetta 2 runtime, Feb 2021.

[57] 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.

[58] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying arm concurrency: Multicopy-atomic axiomatic and operational models for armv8. *Proceedings of the ACM on Programming Languages*, 2(POPL), December 2017.

[59] RIKEN Center for Computational Science (R-CCS), Japan. Fugaku supercomputer. https://www.r-ccs.riken.jp/en/fugaku/project, 2020. Accessed: 2022-07-06.

[60] Rodrigo C. O. Rocha, Dennis Sprokholt, Martin Fink, Redha Gouicem, Tom Spink, Soham Chakraborty, and Pramod Bhatotia. Lasagne: A static binary translator for weak memory model architectures. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 888–902, New York, NY, USA, 2022. Association for Computing Machinery.

[61] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications*, pages 62–71, 2009.

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

[63] Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3):379–405, May 2006.

[64] Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd Millstein, and Madanlal Musuvathi. End-to-end sequential consistency. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 524–535. IEEE, 2012.

[65] Tom Spink, Harry Wagstaff, and Björn Franke. A retargetable System-Level DBT hypervisor. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 505–520, Renton, WA, July 2019. USENIX Association.

[66] Yulei Sui and Jingling Xue. Source Code Analysis with Static Value-Flow, 2016. Accessed: 2022-10-03.

[67] Techcrunch.com. Microsoft updates its Arm-based Surface Pro X tablet with a faster CPU. https://techcrunch.com/2020/10/01/microsoft-updates-its-arm-based-surface-pro-x-tablet-with-a-faster-cpu/, 2020. Accessed: 2022-07-06.

[68] The Guardian. 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, 2020. Accessed: 2022-07-06.

[69] Zhaoguo Wang, Ran Liu, Yufei Chen, Xi Wu, Haibo Chen, Weihua Zhang, and Binyu Zang. Coremu: A scalable and portable parallel full-system emulator. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, page 213–222, New York, NY, USA, 2011. Association for Computing Machinery.

[70] Pan Xinhui. 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, Jun 3, 2016. Accessed: 2022-07-06.

[71] S. Bharadwaj Yadavalli and Aaron Smith. Raising binaries to llvm ir with mctoll (wip paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2019, page 213–218, New York, NY, USA, 2019. Association for Computing Machinery.

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

Received 2022-07-07; accepted 2022-09-22