Will My Program Break on This Faulty Processor?

Formal Analysis of Hardware Fault Activations in Concurrent Embedded Software

LEVENTE BAJCZI, Fault Tolerant Systems Research Group, Department of Measurement and Information Systems, Budapest University of Technology and Economics, Hungary
ANDRÁS VÖRÖS, Fault Tolerant Systems Research Group, Department of Measurement and Information Systems, Budapest University of Technology and Economics, Hungary and MTA-BME Lendület Cyber-physical Systems Research Group, Hungary
VINCE MOLNÁR, Fault Tolerant Systems Research Group, Department of Measurement and Information Systems, Budapest University of Technology and Economics, Hungary and MTA-BME Lendület Cyber-physical Systems Research Group, Hungary

Formal verification is approaching a point where it will be reliably applicable to embedded software. Even though formal verification can efficiently analyze multi-threaded applications, multi-core processors are often considered too dangerous to use in critical systems, despite the many benefits they can offer. One reason is the advanced memory consistency model of such CPUs. Nowadays, most software verifiers assume strict sequential consistency, which is also the naïve view of programmers. Modern multi-core processors, however, rarely guarantee this assumption by default. In addition, complex processor architectures may easily contain design faults. Thanks to the recent advances in hardware verification, these faults are increasingly visible and can be detected even in existing processors, giving an opportunity to compensate for the problem in software. In this paper, we propose a generic approach to consider inconsistent behavior of the hardware in the analysis of software. Our approach is based on formal methods and can be used to detect the activation of existing hardware faults on the application level and facilitate their mitigation in software. The approach relies heavily on recent results of model checking and hardware verification and offers new, integrative research directions. We propose a partial solution based on existing model checking tools to demonstrate feasibility and evaluate their performance in this context.

CCS Concepts: • **Software and its engineering → Formal software verification**; • **Computer systems organization →** Processors and memory architectures.

Additional Key Words and Phrases: fault, analysis, memory consistency model, concurrent, litmus test

This article appears as part of the ESWEEK-TECS special issue and was presented at the International Conference on Embedded Software (EMSOFT) 2019.
Authors' addresses: Levente Bajczi, Fault Tolerant Systems Research Group, Department of Measurement and Information Systems, Budapest University of Technology and Economics, P.O. Box 91, Budapest, Hungary, H-1521; András Vörös, Fault Tolerant Systems Research Group, Department of Measurement and Information Systems, Budapest University of Technology and Economics, P.O. Box 91, Budapest, Hungary, H-1521 , MTA-BME Lendület Cyber-physical Systems Research Group, Budapest, Hungary, vori@mit.bme.hu; Vince Molnár, Fault Tolerant Systems Research Group, Department of Measurement and Information Systems, Budapest University of Technology and Economics, P.O. Box 91, Budapest, Hungary, H-1521 , MTA-BME Lendület Cyber-physical Systems Research Group, Budapest, Hungary, molnarv@mit.bme.hu.

1539-9087/2019/10-ART111 $15.00
https://doi.org/10.1145/1122445.1122456

ACM Reference Format:
Levente Bajczi, András Vörös, and Vince Molnár. 2019. Will My Program Break on This Faulty Processor?: Formal Analysis of Hardware Fault Activations in Concurrent Embedded Software. *ACM Trans. Embedd. Comput. Syst.* 18, 4, Article 111 (October 2019), 22 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Concurrency in critical embedded systems is a field under intensive research. Concurrent programs are hard to write correctly, and running them on multi-core processors is also often considered dangerous because of their complex memory model and recently revealed design flaws[28].

From the software side, formal verification specialized to concurrent programs is steadily improving (see e.g., [19]) and tools compete annually in the various tracks of the competition on software verification[1]. From the hardware side, recent results in the verification of memory consistency models [28] and compilers [25] open new windows in the deployment of correct software on correct hardware. All these advancements bring software and hardware verification closer and closer, but full-stack verification of embedded computer systems will probably remain too complex for some time.

The premise of this paper is that the most recent tools in hardware, and especially memory consistency model verification carries the ability to detect many errors during the design of new, but also existing microprocessors. This is increasingly important in the age where custom silicon appliances could potentially become the future of embedded applications due to open-source architectures and solutions such as RISC-V[2].

Assuming we are able to detect design flaws retrospectively in existing processors (which happens more and more often, see e.g., [18]), there may be a need to decide if a given flaw will ever be activated in an application or not. With this information, it would be easier to decide about a potential recall of the hardware or even mitigate the problem from software.

Therefore we propose a new kind of problem in the intersection of software and hardware verification. An exact solution needs to consider the behavior of the program, but also the execution semantics of the hardware to a limited degree. In our opinion, much of the necessary methodology is already known and published (e.g., in [19] and [28]), but their combination is not trivial.

In this paper, we formalize the proposed problem building on the existing methodology and outline a possible exact solution. We also consider some approximative solutions and design one of them based on traditional model checkers to provide a proof of concept. Finally, we evaluate two implementations with two significantly different model checkers to demonstrate feasibility and see how traditional model checking algorithms handle such problems.

2 BACKGROUND

This section introduces the key concepts in formal verification (of software), computer architectures and memory consistency models to establish the foundations of our work in both the hardware and the software world.

2.1 Formal Verification

Formal methods in computer science provide a methodology for using mathematical techniques in the specification, design and verification of computer systems [10]. Formal verification deals with the problem of proving or refuting if a formal system model satisfies its formal specification. Formal verification techniques differ from traditional verification (such as testing) in that they aim to unambiguously prove properties, often by considering every possible behavior of the system.

[1] https://sv-comp.sosy-lab.org/
[2] https://www.sifive.com/

A wide-spread technique for automatic formal verification is model checking [10]. Given a formal specification, model checking explores the set of possible behaviors of the model – the *state space* – to check conformance. Model checking may generate a *witness*, which is usually an execution trace that demonstrates how the property is satisfied/violated.

2.1.1 Partial Order Reduction. Partial order reduction (POR) is a technique usually used with explicit state space enumeration [15]. In concurrent systems, the combinatorial explosion of the state space is often caused by the arbitrary interleaving of independent behaviors. The goal of POR is to eliminate the redundant interleavings by keeping track of dependencies in the model and prioritizing independent transitions [12].

2.1.2 Symbolic Model Checking with Decision Diagrams. Many symbolic model checking techniques use decision diagrams to compactly store the state space [8]. A decision diagram is similar to a decision tree, but isomorphic sub-trees are merged. The decision encoded in the diagram is whether a given state vector is part of a set (the state space) or not. This approach scales well with concurrent, asynchronous systems, because the similarity of states can be exploited by the diagram [7].

2.2 Computer Architecture

Computer architectures, more precisely called *instruction set architectures* (ISA) are abstract models of a processing unit, defining the interface between software and hardware. On the software side, an ISA defines the available instruction primitives and their semantics, while it is also the specification for the hardware implementation.

An ISA for multi-core processors needs to specify how memory operations of a processing unit will be observable to other units. This is non-trivial because modern processors are usually implemented with various optimizations, such as caches and read/write-buffers and often employ out-of-order execution, meaning instructions of a program are executed in a data-driven manner instead of the order specified by the program. The part of the ISA governing these issues is called the *memory consistency model* (MCM).

2.2.1 Memory Consistency Model. The main role of the MCM is to define *ordering rules*. Even though the computing units in a multi-core processor are executing in parallel, access of the (shared) system memory will always be serialized by the memory controller and the corresponding bus – this serialization must be done according to the ordering rules.

A memory operation can either be a *store* (write), *load* (read) or a special *read-modify-write* (RMW) operation. A write is considered successful when all memory regions of the location are updated with the new value (main system memory and cache(s), when used), while a read is successful when the return value is bound (cannot change later on during the operation) [13]. A RMW can be considered as a read and a write executed atomically in one operation.

Ordering rules might be specified by using *litmus tests* (both Intel and AMD do this [6, 23]), which are small concurrent programs containing memory operations executed on different computing units (threads), along with a specification of *forbidden outcomes*. Outcomes are always defined as values of local non-shared memory or registers (when applicable) after storing and loading data to and from shared memory regions. Forbidden outcomes therefore specify what results *may not* occur when executing the test program by observing and constraining how memory operations can affect each other.

2.2.2 Sequential Consistency. The baseline for the ordering rules is the *program order*, the order in which the program specifies the memory operations. Every processor must ensure that a single thread running on a single computing unit observes its own effect as if the instructions were executed in program order, but this may be relaxed for other threads running on other computing

```
1  A = 1;          1  B = 1;
2  u = A;          2  v = B;
3  w = B;          3  x = A;
```

Fig. 1. Litmus test for the TSO model. The outcome $(u, v, w, x) = (1, 1, 0, 0)$ is forbidden by SC but allowed by TSO.

units. The most conservative approach, which specifies that instructions of every thread should be observable by every thread according to program order, is called *sequential consistency* (SC).

SC corresponds to a very simple computer architecture where every memory operation goes directly to the memory (without caches or buffers) in program order, and the memory serves the requests one-by-one. Due to its natural simplicity, SC is the consistency model implicitly assumed by programmers and also most formal verification algorithms – we could call it a *programmer-centric* memory model [13].

Modern processors, however, rarely guarantee SC by default. Since the memory is usually orders of magnitude slower than the processor, various optimizations are employed to compensate this difference, some of which require the relaxation of SC [13].

2.2.3 Relaxed Memory Models. There are generally 6 possible relaxations of SC: reordering *Write-to-Read*, *Write-to-Write*, *Read-to-Write* and *Read-to-Read* when these instructions apply to different addresses, as well as early reading of own or other unit's write [13]. Introducing a *write buffer*, for example, will cause a possible *Write-to-Read* reordering, i.e., reads from different addresses may overtake writes, potentially reading a value that should have been overwritten in SC. An important MCM with this relaxation is *total store ordering* (TSO), which was proposed for the SPARC V8 architecture[27]. A litmus test illustrating the relaxed behavior is given in Figure 1.

Nowadays, the most common MCM is *weak ordering* (WO), which employs all 6 relaxations [9, 11]. This allows aggressive optimizations in the processor implementation, offering a huge speedup compared to SC. Since WO does not preserve the order of memory operations on different addresses and also allows the early reading of writes, the only way to reliably synchronize between cores is to introduce a special instruction, often called a *memory barrier* or *fence*. These denote points in the execution where all memory operations issued before the synchronization must be finished completely before further operations are to be issued.

When using relaxed memory models, many well-known mutual exclusion protocols will fail: e.g., Peterson's algorithm [14], used as a running example in this paper, is known to work only with SC. The C language used to be defined over SC, but many modern languages (such as C11 and Java) are now defined over WO. This requires additional care from the programmer, but also provides a substantial speedup.

2.2.4 Memory Models of Programming Languages. Since programming languages are higher-level abstractions than the ISA, most of them define their own memory model that is implemented for each ISA through the compiler [25]. Programmers can then specify the minimum guarantee they expect from the platform, tuning the execution of their memory operations between *optimizable* and *strictly sequential* depending on the needs of the application.

In C/C++11, the ordering modes for a single store or load include *seq_cst* and *relaxed*, which correspond to SC and WO [17]. There are also less relaxed modes such as *acquire* and *release*, which provide a middle ground. A synchronization occurs when thread A issues a *release store* and thread B issues an *acquire load* that reads the value stored by thread A. Accordingly, operations issued before this point in thread A will be visible to B, while operations of B issued after this point

```
1  C corr_R_relaxed_relaxed_W_relaxed_relaxed
2  { [x] = 0; }
3  /* Assigning different values to 'x' */
4  P0 (atomic_int* x) {
5    STORE(x, 1, memory_order_relaxed);
6    STORE(x, 2, memory_order_relaxed);
7  }
8  /* Reading back the value of 'x' two times */
9  P1 (atomic_int* x) {
10   int r1 = LOAD(x, memory_order_relaxed);
11   int r2 = LOAD(x, memory_order_relaxed);
12 }
13 /* Forbidden outcome: r1 = 2 and r2 = 1 */
14 exists (1:r1 = 2 /\ 1:r2 = 1)
```

Fig. 2. The corr_R_relaxed_relaxed_W_relaxed_relaxed litmus test using the macros defined in Figure 3.

will *not* be visible to A. Furthermore, an explicit memory fence instruction is provided to force synchronization (the other modes are also often implemented in terms of fence instructions).

Programming language-level memory models enable the formal analysis of a concurrent program without considering compiler, operating system and hardware specifics, assuming these are implemented correctly, i.e., they provide the guarantees defined by the programming language-level memory model.

2.3 Formal Verification of Memory Models

With the increasing complexity of ISAs and hardware implementations, there is more and more emphasis on the formal verification of hardware designs. This is motivated by the exponentially increasing cost of detecting a design flaw, and also by the fact that there has been a number of occasions when such a flaw was not detected before the commercial release of the affected product.

Regarding MCMs, a famous example that we use in the paper is the so-called *ARM Read-Read Hazard*[4]. In this case, two successive reads to the same address could reorder themselves in a way disallowed by the specification[5], which specifies that every processing core must observe its own memory operations in program order, and no other thread might observe them in a different order. A litmus test that can reveal the problem is shown in Figure 2.

Formal verification of memory models includes at least three stages due to the various abstraction layers. On the lowest level, the hardware implementation must be checked against the ISA to manufacture correct chips (see Section 3.1 and [28]). On the middle level, compiler mappings from programming language memory models to different ISAs should also be verified to ensure guarantees offered to the programmer (see Section 3.2 and [25]), while on the highest level relaxed memory models should also be considered in the formal verification of software and algorithms (see e.g., Section 3.4).

2.4 Execution Graphs

An efficient way to describe executions of parallel programs is to use *execution graphs* [19]. An execution graph is a directed acyclic graph that defines a partial order of instances of operations (with concrete parameters), encoding explicit dependencies but not specifying a total execution order. Typically, these dependencies will come from either the program order (*po*) or memory

operations. An advantage of execution graphs over the interleaving semantics of model checking is that the source of data for read operations (i.e., the write operation that produced the data) can also be denoted by a dependency between the memory operations ("read-from", *rf*).

This approach can also be used to describe the behavior of hardware pipelines implementing a memory model [21]. In this case, the graph is often called a "happens before" graph as it is used to reconstruct a possible set of behaviors leading to an observed outcome (i.e., operations had to *happen before* other operations to produce the output). Many types of dependencies can introduce a "happens before" arc, including the two mentioned above [2]. Note that a valid "happens before" graph is always acyclic – if there is a cycle, the graph cannot be an execution graph because there is no legal total ordering of the operations (nothing can happen before itself).

In this paper, we use execution graphs for both purposes (see e.g., Figures 5 and 7), building on the related work presented in the next section.

3 RELATED WORK

To the best of our knowledge, the approach proposed in this paper is the first of its kind. There are, however, concepts and research that are closely related to our methodology around the boundaries of the following three topics: hardware verification, compiler verification and software verification.

3.1 MCM Verification with TriCheck

In [28], the authors introduce a tool called *TriCheck* to demonstrate how MCM inconsistencies can be found, applied to the open-source RISC-V architecture. TriCheck builds on three other tools to verify memory consistency models: *PipeCheck* [21] to build and analyze a "microarchitecturally happens before" graph describing the dependencies between stages of the pipeline executing different memory operations; CCICheck [24], which extends the methodology to verify the *coherence-consistency interface* related to caching; and COATCheck [22] that further extends the verification to consider *virtual address memory consistency*.

The goal of the tool is to formally prove that forbidden outcomes of litmus tests can indeed never occur in a specific hardware implementation. The main idea is that any outcome that does not introduce a cyclic dependency between the operations has an execution graph and therefore may actually be observable.

In a different project [3], a complementary tool[3] called *litmus7* has been introduced that can be used to actually observe these executions on real hardware. By applying different timings, the tool can make these execution corner-cases appear more frequently, often raising the number of forbidden results to a few dozens in every billion executions.

TriCheck was the main inspiration for this work. Even though it aims to help hardware manufacturers to avoid errors in the implementation, there are well-known cases when a design flaw was revealed well after the release of the CPU, such as the already mentioned ARM Read-Read Hazard[5], or the more famous Spectre attack[18] (although Spectre is not in the scope of our work). TriCheck can be used to verify already released processors to possibly reveal rare but critical faults in embedded systems to facilitate their repair. Whether or not a design flaw will be activated in a certain application is the main focus of our work.

Furthermore, the methodology of building dependency graphs to define the partial order of execution steps can be generalized to give an exact – although not very feasible – solution to the problem we introduce (see Section 5).

[3] http://diy.inria.fr/doc/litmus.html

3.2 Compiler Verification for Weak Memory Models

As mentioned in Section 2.2.4, memory models of programming languages are mapped to memory models of an ISA by a compiler. From the correct software–correct compiler–correct hardware triple, [25] addresses the correct compiler problem. Using a new intermediate weak memory model (IMM), they modularize the proofs of correctness of compilation from concurrent programming languages with WO memory semantics to multi-core architectures. The case study yields the first machine-verified compilation correctness results for models that are weaker than x86-TSO.

3.3 Library Correctness over WO Memory

As in other types of software, concurrent applications also use libraries for basic building blocks. There had been work verifying concurrent libraries over SC, but not much had been known about their correctness over WO. The framework proposed in [26] enables the specification of concurrent libraries declaratively and verifies their behavior with regard to this specification in a compositional way. The framework is suitable for encoding standard memory models such as SC and TSO, as well as the memory model of the C11 language. Applicability has been demonstrated on libraries implementing locks, exchangers, queues and stacks.

This work and the next one are similar to our work in that they seek to consider the memory model in the verification of concurrent software.

3.4 Stateless Model Checking for C/C++

In [19], a novel approach is proposed for the stateless model checking of concurrent programs written in RC11, a "repaired" version of C/C++11 without dependency cycles (introduced in [20]). They use a variant of stateless model checking combined with partial order reduction to generate all consistent *execution graphs* (instead of states, which would imply an interleaving semantics). The significance of the algorithm is that it considers the weak memory model in the verification process. The resulting tool, called RCMC, is shown to scale significantly better than traditional model checkers. In [1], an extension is proposed to consider the release-acquire memory model as well.

In addition to the relevance of the goal, this approach may turn out to be the closest to how an optimal solution to the problem proposed in this paper would look like.

4 OVERVIEW

Context. Assume there is a multi-core processor with a relaxed MCM specified by litmus tests as seen in Section 2.2.1, as well as a concurrent program in e.g., C/C++11 that is to be executed by this processor. Also, assume that the processor is known to have a design flaw that causes one of the forbidden outcomes of a litmus test to be occasionally observable, but this flaw cannot be repaired practically because the hardware is already manufactured.

Motivation. Because of the enormous cost of repairing or recalling the hardware, we want to check if the program even activates the fault. The fault is activated only if the program can produce the behavior described by the litmus test – in this case, the forbidden output will be observable and may affect the correctness of the program. Detecting the activation also gives an opportunity to mitigate the problem from software.

Challenges. Detecting if a program will activate the fault is not possible without analyzing its dynamic behavior, because the trigger is two or more *simultaneously executed* sequences of memory operations. Dynamic detection, however, means that the patterns described by the litmus test(s) must be mapped to the *state space* of the program, which is in itself hard to compute. Furthermore,

```
1  // Header
2  #include <stdatomic.h>
3  #define STORE(adr, val, ord) atomic_store_explicit(adr, val, ord)
4  #define LOAD(adr, ord) atomic_load_explicit(adr, ord)
5
6  // Global variables
7  _Atomic int w = 0;
8  _Atomic int flag[2] = {0, 0};
9  _Atomic int turn = 0;
10
11 // Code of Thread 0
12 void thread0() {
13   STORE(&flag[0], 1, memory_order_relaxed);
14   STORE(&turn, 1, memory_order_relaxed);
15   int m_flag, m_turn;
16   do {
17     m_flag = LOAD(&flag[1], memory_order_relaxed);
18     m_turn = LOAD(&turn, memory_order_relaxed);
19   } while (m_flag == 1 && m_turn == 1);
20   // critical section
21   STORE(&w, 1, memory_order_relaxed);
22   STORE(&w, 2, memory_order_relaxed);
23   // end of critical section
24   STORE(&flag[0], 0, memory_order_relaxed);
25 }
26
27 // Code of Thread 1
28 void thread1() {
29   STORE(&flag[1], 1, memory_order_relaxed);
30   STORE(&turn, 0, memory_order_relaxed);
31   int m_flag, m_turn;
32   do {
33     m_flag = LOAD(&flag[0], memory_order_relaxed);
34     m_turn = LOAD(&turn, memory_order_relaxed);
35   } while (m_flag == 1 && m_turn == 0);
36   // critical section
37   int r1 = LOAD(&w, memory_order_relaxed);
38   int r2 = LOAD(&w, memory_order_relaxed);
39   // end of critical section
40   STORE(&flag[1], 0, memory_order_relaxed);
41 }
```

Fig. 3. Running example: Peterson's mutual exclusion algorithm with a critical section matching the corr_R_relaxed_relaxed_W_relaxed_relaxed litmus test in Figure 2. Note that if mutual exclusion works, these instructions will not match the litmus test, as they cannot be executed in parallel.

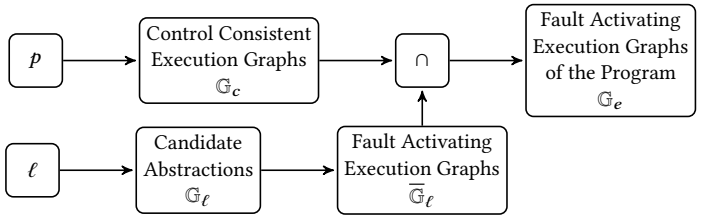

Fig. 4. Overview of the formal problem definition.

the mapping problem is parametric, because any subset of the program's threads could match the threads of the litmus test, with any subset of shared variables. Last, but not least, the computation of the state space must take the memory model into account to yield a complete solution.

Approach. First, we formally define the goal as a decision problem, then outline a precise theoretical solution that considers the most recent results in the related work. Since currently no tool exists that could solve the problem in a precise way, we propose an underapproximation built on existing model checkers and model transformations to demonstrate the approach in practice and get an initial picture about how the existing solvers can handle this class of problems.

Running Example. We will use Peterson's algorithm for mutual exclusion [14] to illustrate the concepts of our approach. This algorithm is known to require sequential consistency to guarantee its properties. The implementation shown in Figure 3 uses relaxed memory operations, which will *break the algorithm* on relaxed memory models. In the critical section, the instructions are from the threads of the corr_R_relaxed_relaxed_W_relaxed_relaxed litmus test. If mutual exclusion is not guaranteed, these instructions can be executed in parallel. We will use this to demonstrate the capabilities and limitations of the proposed underapproximation.

5 FORMAL PROBLEM STATEMENT

Informally, our goal is to detect the activation of a hardware fault of a given processor in a given concurrent program p. We will use a litmus test ℓ to characterize the situation when the fault activates. ℓ is selected by a specialized MCM verification tool such as TriCheck (see Section 3.1) by proving that a forbidden outcome is allowed by the concrete hardware implementation. We assume there is such a litmus test for every class of situations when the fault activates because manufacturers should provide a litmus test for every guarantee they give [16] and there are complementary approaches to synthesize a comprehensive litmus test suite [23]. Note that the following considerations apply to the theoretical problem definition only and do not describe a concrete (let alone efficient) algorithm.

The overview of the following ideas is as follows (see Figure 4 for an illustration). Starting from p, we generate execution graphs $\mathbb{G}_c$ that could occur on an arbitrary (faulty) memory controller. These graphs capture the control semantics of the program but contain inconsistent memory operations. Next, we take the execution graph of ℓ corresponding to the forbidden outcome and extend it with every possible (transitive) dependency between its operations that still enables the fault to activate ($\mathbb{G}_\ell$). We further extend these graphs with arbitrary operations and control dependencies, keeping the consistency of memory operations apart from those of the litmus test but ignoring actual control flow and consistent control dependencies ($\overline{\mathbb{G}}_\ell$). The intersection of these two sets of graphs – if non-empty – contains the execution graphs of p that activate the fault found by ℓ: $\mathbb{G}_c \cap \overline{\mathbb{G}}_\ell$.

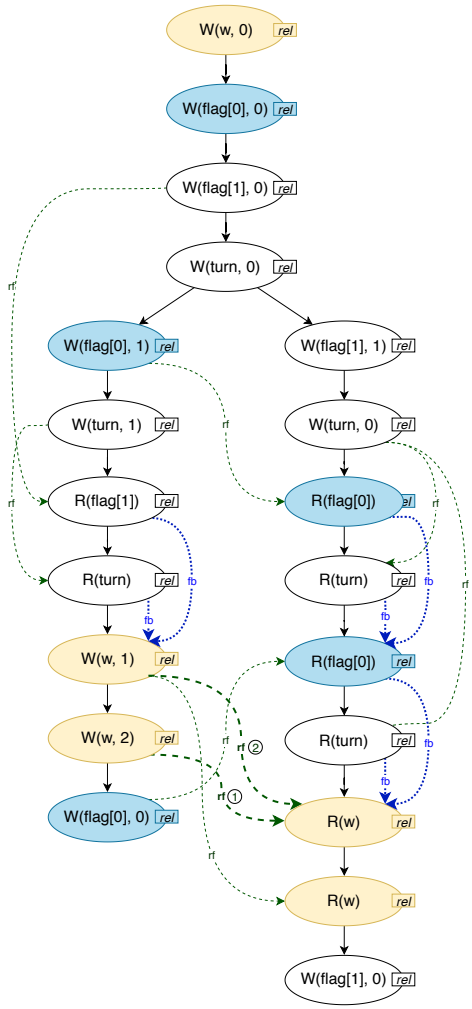

Fig. 5. Two variants of an execution graph of Peterson's algorithm as presented in Figure 3. This graph shows a situation where mutual exclusion is achieved.

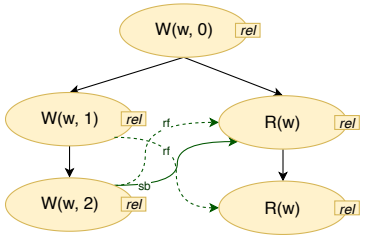

Fig. 6. Forbidden but observable execution graph of the corr_R_relaxed_relaxed_W_relaxed_relaxed litmus test over the w memory location, extended by an *sb* dependency.

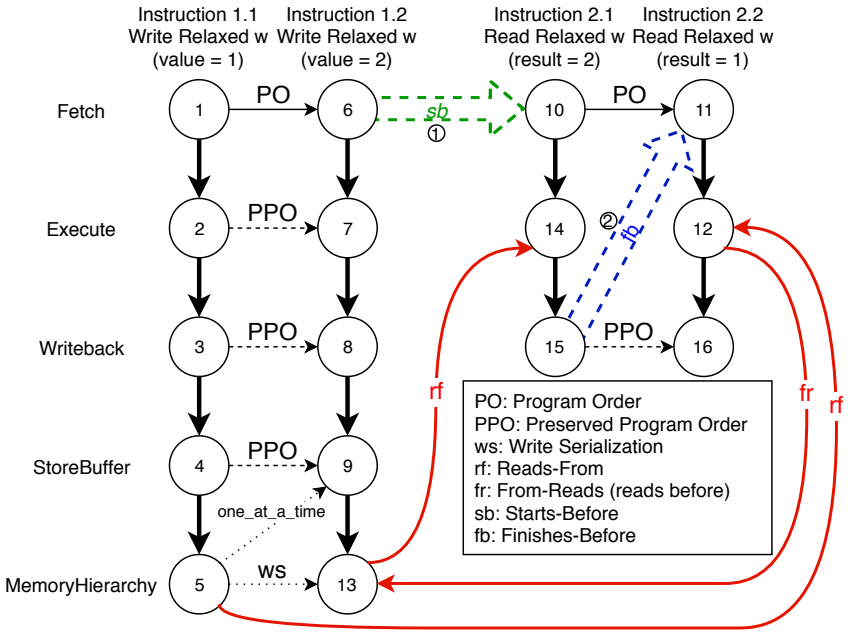

Fig. 7. The "happens before" graph of the corr_R_relaxed_relaxed_W_relaxed_relaxed litmus test on a sample hardware, created by TriCheck [28]. Write serialization denotes the execution order in the memory controller. Arrows marked with ① and ② belong to an example and are not part of the original graph. Note the lack of cycles which makes a forbidden output observable: the numbering of the nodes gives a topological order with no "backward" arcs, which is a feasible execution. Also note that this outcome would be impossible had the hardware correctly implemented a PPO dependency between nodes 14 and 12 as it would introduce a cycle. This is also the case with the additional *fb* dependency marked with ②, but not with the *sb* relation marked with ①.

5.1 Control Consistent Execution Graphs

A trivial case is when p contains ℓ directly as a concurrently executable subprogram. More interesting cases, however, include the execution of additional operations between and around the instructions of the litmus test. These additional instructions cannot introduce additional observable outcomes, but they may remove some by adding more dependencies – potentially (but not necessarily) removing the observable forbidden outcome (e.g., by serializing the parallel threads with some form of synchronization). To characterize this, we introduce two additional types of dependencies between memory operations.

Definition 1. *An operation A* "starts before" *(*sb*) operation B if A must always be issued before B. An operation A* "finishes before" *(*fb*) operation B if 1) A is a write operation and the stored value must be visible to all threads in the whole memory hierarchy (including caches) or 2) A is a read operation and the final loaded value must be available in the calling thread* before *B is issued.*

A *fb* dependency can denote multiple situations: *1)* a read operation reads the value stored in a write operation (*rf*), *2)* a different result of a read would divert the control flow and prevent the execution of another memory operation (control dependency), *3)* a *sequential* memory operation always finishes before any operation can starts that are after it in program order (on the same thread) and starts only after all operations finished that are before it in program order, *4)* a *release*

store starts only after all operations finished that are before it in program order and *5)* an *acquire load* always finishes before any operations starts that are after it in program order (see [2] for the justification of *3–5*).[4] This dependency is transitive, but only through other *fb* dependencies. A *sb* dependency can model when the logical order of instructions in an execution implies an ordering between the issuing of operations (i.e., program order and *fb* dependencies, transitively).

To reason about potential executions of a concurrent program (either a litmus test or the program under analysis), we will use execution graphs with "reads-from" (*rf*) dependencies between read and write operations and extended with *sb* and *fb* dependencies. In addition to the *consistent execution graphs* used in [19], we will define *control consistent execution graphs* to model behaviors with arbitrary memory consistency errors (instead of only the errors of the specific processor under consideration).

Definition 2. *A* control consistent execution graph G_c *is an execution graph of program p (see [19]) in which* rf *arcs are unrestricted, i.e., a read operation may read from* any *write operation in the graph as long as the stored and loaded values are the same. Note that the control flow must obey the loaded values and the control structures of p (i.e., we assume that the processor is flawless apart from the memory controller).*

Two control consistent execution graph of Peterson's algorithms as presented in Figure 3 can be seen on Figure 5. The numbered *rf* dependencies denote the two cases: with ① the graph is not consistent but control consistent, with ② it is also consistent.

5.2 Fault Activating Execution Graphs

A litmus test with an observable forbidden outcome comes with a "happens before" graph H_ℓ that denotes the various dependencies between the pipeline stages of different memory operations. As seen before, if this graph is acyclic, it means that the forbidden outcome may be observable. As an example, see Figure 7 that denotes a situation in which the ARM Read-Read Hazard is observable on a processor. On this graph, a *sb* dependency applies to the first stage of each instruction, while a *fb* dependency orders the last stage of the first instruction before the first stage of the second instruction.

If we have the set of all control consistent executions $\mathbb{G}_c$ of p, we can check whether any of them is "similar" to the execution graph G_ℓ of ℓ with the forbidden (but observable) outcome. To do this, we generate a set of execution graphs $\mathbb{G}_\ell$ derived from G_ℓ by adding *sb* and *fb* dependencies in every possible way both to G_ℓ and H_ℓ as specified above, such that H_ℓ remains acyclic. These will be the candidate abstractions of execution graphs in $\mathbb{G}_c$.

The last thing to check is if any of these candidates can be extended into a "quasi-consistent" execution graph in $\mathbb{G}_c$. A "quasi-consistent" execution graph is an execution graph which is consistent in the sense defined in [19] (there exists a program over WO that can produce the described executions) *except* for the *rf* arcs originally present in execution graphs in $\mathbb{G}_\ell$. The extensions, denoted by $\overline{\mathbb{G}}_\ell$ and called fault activating execution graphs can be realized by iteratively applying the following transformations on elements of $\mathbb{G}_\ell$ in a way that no new (transitive) *sb* and *fb* dependencies appear between the original memory operations of the litmus test:

(1) Insert any new node X before the first existing node A (such that X is *po* before A, denoted by $X \rightarrow A$).

[4] Assuming that these situations indeed lead to a *fb* relation means that the hardware is correct, which may seem contradictory. Nevertheless, the specific hardware fault that we are looking for will be considered separately, and there is always a "first error" that can be found without assuming a faulty hardware.

(2) Insert any new node X after the last existing node A (such that $A \rightarrow X$), or insert a new node X after an existing non-terminal node A followed by any node B (such that $A \rightarrow B$ and $A \rightarrow X$ but $B \nrightarrow X$ and $X \nrightarrow B$).
(3) Insert any new node X between an existing node A and a set of existing nodes $\mathcal{B}$ in which for every node $B \in \mathcal{B}$ we have $A \rightarrow B$ (such that $A \rightarrow X$ and $X \rightarrow B$ for every $B \in \mathcal{B}$).
(4) Insert a *fb* dependency between any two nodes without introducing a cycle to denote control dependency.
(5) Insert a *rf* arc consistently between a write and a read that does not have an incoming *rf* arc.

Figure 6 shows the forbidden but observable execution graph of the litmus test from Figure 2 (corr_R_relaxed_relaxed_W_relaxed_relaxed) extended with a *sb* dependency (an element of $\overline{\mathbb{G}}_\ell$). This dependency does not prevent the forbidden outcome, as H_ℓ extended with it (Figure 7 with the *sb* arc marked with ①) is still acyclic. Also note that the graph can be extended with the transformations above to get the control consistent execution graph in Figure 5.

5.3 Fault Activations of the Program

The intersection of the control consistent execution graphs $\mathbb{G}_c$ of p and the fault activating execution graphs $\overline{\mathbb{G}}_\ell$ will be denoted by $\mathbb{G}_e$. This set contains execution graphs that are consistent with the control flow and dependencies specified by p (because they are in $\mathbb{G}_c$) but also consistent with the WO memory model apart from the fault described by ℓ (because they are in $\overline{\mathbb{G}}_\ell$), therefore all overapproximations introduced in the definitions above are excluded. Furthermore, $\mathbb{G}_e$ contains every execution graph of p that activates the hardware fault described by ℓ. Based on this, the problem statement is as follows.

Problem Statement 1. *Given a concurrent program p and a litmus test ℓ, let L be the set of execution graphs of ℓ that correspond to the forbidden outcome with any set of shared memory locations in p. For each $G_\ell \in L$, let $\overline{\mathbb{G}}_\ell$ be the set of fault activating execution graphs as defined above and let $\mathbb{G}_c$ be the set of control consistent execution graphs of p. Then the question of having an activation of the fault described by ℓ in p or not is equivalent to deciding $\mathbb{G}_e = \mathbb{G}_c \cap \overline{\mathbb{G}}_\ell \stackrel{?}{=} \emptyset$.*

According to this definition, Figure 5 shows an execution graph that describes an activation of the ARM Read-Read Hazard over variable w.

5.4 Outline of an Exact Solution

The definition of Problem 1 includes the generation of multiple infinite sets of execution graphs, which is definitely not feasible in practice. An exact solution should efficiently generate candidate execution graphs that may be in the intersection and prove that it is an element of both sets. In this section, we outline a solution that could build on existing, although rather new techniques to generate and check such candidates.

The key observation is that every execution graph in $\mathbb{G}_e$ is *almost* a consistent execution graph of p, the only inconsistency is exactly the one described by the execution graph of the litmus test that corresponds to the forbidden outcome. The algorithm presented in [19] (the efficient RCMC model checking algorithm) is capable of enumerating all *consistent* execution graphs of p and it can be extended to keep track of *fb* dependencies introduced by control structures as well. It can also do this with litmus tests, and although these consistent execution graphs will not contain the one leading to the forbidden outcome, we may use them to detect when p behaves like the litmus test. This can be done by matching the execution graphs of the ℓ on the execution graph of p as graph patterns, allowing a "program order" relation to match on a path of "program order" arcs.

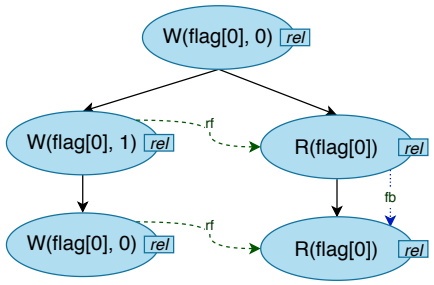

Fig. 8. A potential execution graph of the corr_R_relaxed_relaxed_W_relaxed_relaxed litmus test over the `flag[0]` memory location.

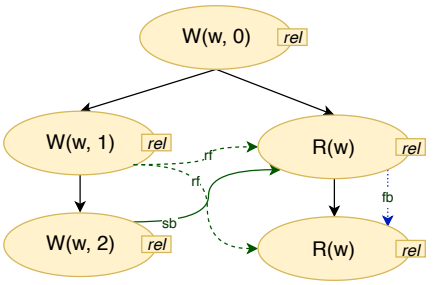

Fig. 9. A potential execution graph of the corr_R_relaxed_relaxed_W_relaxed_relaxed litmus test over the `w` memory location.

Once we have a match (denote it by G), we have to trace back the transitive *sb* and *fb* relations induced by *po*, *rf* and *fb* dependencies (introduced by control structures and other memory operations) into the "microarchitecturally happens before" graph H_ℓ of the forbidden outcome. If the additional dependencies introduce a cycle, this execution may not activate the fault because H_ℓ is not an execution graph. Otherwise, the matched execution graph may describe a situation when the hardware fault activates. To verify this, we change the *rf* dependencies between the matched memory operations to correspond to the forbidden outcome, obtaining G_e, which is an element of $\overline{\mathbb{G}}_\ell$ (because it is consistent apart from the activated fault and a valid extension of the forbidden execution graph of the litmus test).

The last thing to check is whether $G_e \in \mathbb{G}_c$, i.e., if it is consistent with the control flow defined by p for the read values. This can be done either by the extended RCMC algorithm or by simulating p with mocked read operations returning the values specified by G_c – if the simulation can yield the same control flow as the original execution graph, we have a fault activating program execution. On the other hand, the fault activation may cause the program to read values that divert the control flow – yielding a new control flow graph. In this case, G_e was not in $\mathbb{G}_c$, but the new graph G'_e is. One last time, we have to trace back the dependencies of G'_e into H_ℓ to see if there is a cycle (in which case there is no fault activation), but if not, we have that $G_e \in \mathbb{G}_c$ and $G'_e \in \overline{\mathbb{G}}_\ell$, so G'_e is a fault activation of program p.

In the running example, there are two matches in the sample execution graph in Figue 5 (interpreted with the consistent *rf* relation marked with ②), denoted by the two colors: Figure 8 shows the one over the `flag[0]` variable and Figure 9 the one over `w`. Both figures show the additional dependencies implied by the execution graph of Peterson's algorithm. Tracing them back to the "happens before" graph in Figure 7, we can see that the match related to `flag[0]` introduces a cycle (the *fb* arc marked with ②), so there will be no fault activations associated with those memory operations, but the other one related to `w` – as seen in the illustration of the problem statement – does not introduce a cycle. When we change the *rf* arcs according to the forbidden but observable outcome, we effectively switch from the *rf* arc marked with ② to the one marked with ①. This execution graph is control consistent because the values read from `w` do not matter in terms of the control flow (as seen in Figure 3).

In order for the above approach to work, there always has to be a consistent execution graph of p which differs from the actual fault activating graph in the *rf* arcs only. We prove that this is the case.

Theorem 5.1. *If a concurrent program p has an execution graph $G_e \in \mathbb{G}_e$, there exists an execution graph G such that G_e and G differs only in* rf *dependencies between operations matched to the litmus test, i.e., for every fault activating execution graph there is an equivalent execution graph where the fault does not activate.*

Proof. We prove by contradiction. Assume there is a p which has an execution graph $G_e \in \mathbb{G}_e$ but no corresponding fault-free execution graph G. This is possible only if the control flow of p depends on the outcome in such a way that the matched operations are issued only in case of the forbidden outcome. Due to causality, the condition responsible for "diverting" execution may not be after the last involved operation. However, if it is before the last operation, its result may not be known during the decision. This would imply that it is possible to decide if an outcome is forbidden based on the results of a subset of the operations, which means the litmus test is not minimal (i.e., it contains unnecessary operations) and there is a minimal litmus test that should be checked instead. Since we assume a set of minimal litmus tests (which is always possible), this is a contradiction (see Section 7 for a discussion of limitations). □

6 APPROXIMATIVE SOLUTIONS

The hardest part of solving Problem 1 is precisely keeping track of and tracing back the additional dependencies to the "microarchitecturally happens before" graph of the hadrware implementation. Therefore, we consider and evaluate approximative solutions which under- or overapproximate the "happens before" graph at the cost of introducing false negatives or false positives. In addition, as there are very few available tools that generate execution graphs of concurrent programs over relaxed memory models, and probably none of them could solve the presented problem out of the box, we implement a partial solution based on traditional model checkers with interleaving semantics (corresponding to SC) and investigate its properties.

6.1 Relaxed Solutions

There are many trivial relaxed solutions such as reporting a potential activation when threads of a program contain the instructions specified in the litmus test (without considering dependencies or control structures). These can be used to quickly determine true negatives before moving on to a more expensive solution.

Although currently not available in any tool, a relaxed solution that is very close to the precise answer would involve a model checker like RCMC (see Section 3.4 and [19]) that can enumerate the consistent execution graphs of a program and a relaxed evaluation of "inserted instructions" that will always report a true activation but maybe also false positives. As mentioned in Section 5.4, detecting an execution graph of a litmus test in a consistent execution graph of the program can be precisely formalized as a graph pattern matching problem, where the pattern comes from the structure of the litmus test (a similar approach for model checkers is presented in Section 6.3). By omitting any of the remaining steps of the outlined exact solution we can report a chance of a fault activation for each match based on the fact that the program indeed performs the memory operations just like the litmus test. Depending on how much of the additional dependencies we consider, we may rule out a varying amount of false positives. This solution may still be useful if the program simply does not use the types of operations involved in the litmus test in that specific way.

6.2 Stricter Solutions

When proving that an activation is indeed present, stricter solutions might be able to provide a proof with fewer resources or simpler tools. Similarly to the relaxed solution outlined in the

previous section, we can define a stricter algorithm by modifying how we approximate cycles in the "happens before" graph. One possibility that is again very close to the precise solution is to report "no cycle" whenever there is any additional dependency between the operations matched by the litmus test, i.e., some control structures, non-relaxed memory operations or operations on an address used in the matching litmus test affect how the threads execute. These situations *might* cause an activation but not always (false negatives), while in every other case the instructions on different threads of the litmus test can be executed in parallel, which corresponds to the execution of the litmus test. In this case, checking the control consistency of the forbidden outcome is not even necessary as we do not consider execution graphs where this might be an issue. This solution could detect when programs execute memory operations in the litmus test in an unconstrained, truly parallel way.

6.3 Partial Solution with Model Checking

In this section, we define a special partial solution that works with traditional model checkers. Although most model checkers assume SC during state space generation, an important advantage is that they are general solvers that allow us to formalize a solution based on out-of-the-box tools. We call this a partial solution because the limitation to SC has serious consequences on completeness.

6.3.1 Problem Statement for Model Checking. The stricter problem we want to solve with the model checking-based approach is as follows.

Problem Statement 2. *Given a concurrent program p and a litmus test ℓ, decide whether p has a state s in the space of sequentially consistent behaviors S_{SC} such that*

(1) *for any instance of ℓ where the memory locations are bound to shared variables of p and threads of the litmus test are bound to threads of p, and*
(2) *any thread t of the litmus test*
(3) *there is a sequence of operations π executable from s s.t.*
   (a) *π contains the operations of t in the program order of ℓ, immediately starting with the first one,*
   (b) *π does not have any operations reading or writing a variable used by the instance of ℓ inserted between operations of the litmus test,*
   (c) *π does not have any control instruction that evaluates any variable that (transitively) got its value from a read operation of ℓ, and*
   (d) *π does not have any sequential operation reading or writing* any *variable apart from those in the litmus test.*

Problem 2 is essentially parametric pattern matching on the state space and is related to the model checking of regular properties [10]. The constraints on π ensure that *in WO* no synchronization between the threads can occur during the execution of a litmus test thread and no *fb* dependency is introduced by control instructions (condition 3c). Furthermore, if there is a state from which every litmus test thread can be executed this way, we have that the threads can run independently. Therefore, no transitive dependencies will appear in the "happens before" graph of the litmus test and the forbidden outcome will be observable.

6.3.2 Matching Litmus Tests in the State Space. To detect a path π in the state space S_{SC} that satisfies the constraint defined in Problem 2, we generate nondeterministic finite automata for each thread of the litmus test. As an example, see Figure 10 which describes a satisfying path for thread `P1` of the litmus test from Figure 2. Notice that the automaton is parametric, i.e., it uses the symbols t to constrain the thread executing the given operation and v for the memory location affected

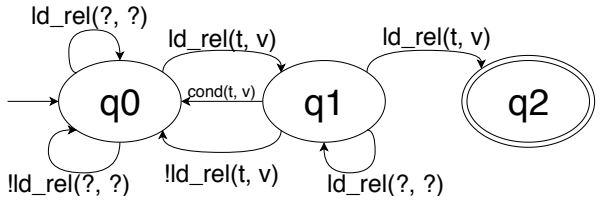

Fig. 10. The automaton for P1 in the litmus test from Figure 2. The informal notation !a matches any memory operation other than a, while *cond(t, v)* means thread t evaluated a condition dependent on the value of v to determine control flow. Parameters are t for thread and v for variable.

```
1  X = 1;
2  X = 2;
```

```
1  r1 = X;
2  if (r1 == 2)
3     r2 = X;
```

Fig. 11. Would this program trigger the Read-Read Hazard? X is a shared variable with initial value of 0.

(there would be more parameters for variables if the litmus test accessed more). This means that $q1 \rightarrow q2$ must read a relaxed store from the same thread and to the same variable as $q0 \rightarrow q1$.

The automata generated this way can be coupled with the model to compute the synchronous product of their state spaces, assuming states of the model are labeled with memory operations and control instructions to be performed next. If the model checker supports properties specified in computational-tree logic, an expression can describe the state s from which there is a satisfying path π for every thread of a given litmus test instance, i.e., every generated automaton can leave its initial state in the next step of some π and reach its accepting state.[5]

6.3.3 Scope of the Partial Solution. Because of the SC semantics assumed by the model checker, the state space S_{SC} we use is just a subset of the real state space S_{WO} that can result from the WO semantics. Therefore the fault activations that are only reachable in S_{WO} will be false negatives. In the running example, the model checking approach would not detect the Read-Read Hazard on w (from Figure 9) because in S_{SC} there is no state from which both threads could start (as the algorithm ensures mutual exclusion in SC). Without the mutual exclusion algorithm, that is, the critical sections alone, this approach would also find and report the problem like the stricter solution outlined in Section 6.2.

7 LIMITATIONS

The most serious limitation of all approaches presented in this paper is that we ignore the effect of other parts of a processor (outside of the MCM). The execution of a program can depend on many internal optimizations, some of which might affect the memory operations issued by a computing unit. One of these is *speculative execution* [18]. Consider the concurrent program in Figure 11 executed on a hardware with the Read-Read Hazard (described by Figure 2). Without speculative execution, the forbidden outcome $(r1, r2) = (2, 1)$ is not observable because that would mean the read operations were reordered (we know that the writes are ordered correctly) – but there can be no second read to reorder if the first one reads 1. With speculative execution, however, there is a chance the processor issues the second read due to branch prediction (this is legal because the

[5]The template for this expression is $EF(\bigwedge_{t \in \ell} EX(started_t \wedge EF accepted_t))$, where $started_t$ and $accepted_t$ mean that the automaton for thread t left its initial state or accepted the path, respectively.

read should be side-effect free), which could indeed trigger the Read-Read Hazard and give us the forbidden outcome (in this case, Theorem 5.1 does not hold).

Furthermore, as highlighted in Section 6, the approximative approaches we proposed and evaluated are either under- or overapproximations and therefore only a best-effort solution aiming to inspire further research towards the precise problem described in Section 5.

A serious threat to the applicability of our approach is that even though we speculate that the set of litmus tests provided by a processor manufacturer covers every interesting situation, this is likely not the case. A promising direction to mitigate this problem is [23], which proposes an approach to synthesize a comprehensive litmus test suite.

There are also various threats to the validity of our measurements, including the implementation of the approach, the proper configuration of the tools we used and other usual threats related to measuring the running time of computer programs.

Last, but not least, the solutions we proposed are not optimal. The model checking of regular properties combined with CTL on concurrent programs is not well-researched, but there are promising techniques that could improve our solutions, such as abstraction-based techniques.

8 EVALUATION

To demonstrate the applicability of the methodology introduced in previous sections we have developed a proof-of-concept implementation of the model checking-based approach. In this section we present an exploratory evaluation of how it scales along the dimensions of the problem and with different model checkers.

The Model Checkers. We implemented the model checking-based approach from Section 6.3 based on two model checkers: *spin*[15] and *nuXmv* [8]. Spin seemed to be a good candidate because of its flexible input language Promela, and also due to dynamic partial order reduction that is usually very efficient for concurrent, asynchronous models. nuXmv, on the other hand, is a symbolic model checker that uses decision diagrams and SAT solving, which is also traditionally suitable for concurrent program-like models, and it offers CTL model checking that enables an easy formalization of the criteria from Problem 2.

The Models. We used simple synthetic input programs, each with $2 \leq t \leq 20$ threads and $5 \leq v \leq 50$ variables, as well as $1 \leq l \leq 10$ litmus tests that have to be matched. The input programs consist of stores and loads in threads, either all of them with sequential or all of them with relaxed ordering. We have experimented with different allocation of the shared variables to the threads and instructions, but will report results only for the randomly assigned models as there was little difference and it is the most realistic.

The Results. Measurements were conducted on a bare-metal server machine rented from the Oracle Cloud (BM.Standard.E2.64), with 64 cores and 512 GB of RAM, running Ubuntu 18.04. We used *benchexec*[6] to measure and limit the resource consumption of each process to 16 GB of memory and 1200s of running time. Scaling of both tools is illustrated in Figure 12, while a comparison of the tools for each problem is shown in Figure 13. We can conclude that nuXmv scaled much better than spin, possibly because partial-order reduction was less applicable due to the automata we used as a regular property. In the three dimensions of the problem, the number of variables had little effect on the running time. The number of threads and litmus test, however, did affect performance in various degrees. While spin shows super-exponential scaling in both dimensions, nuXmv scales nearly or even below exponentially.

[6]https://github.com/sosy-lab/benchexec

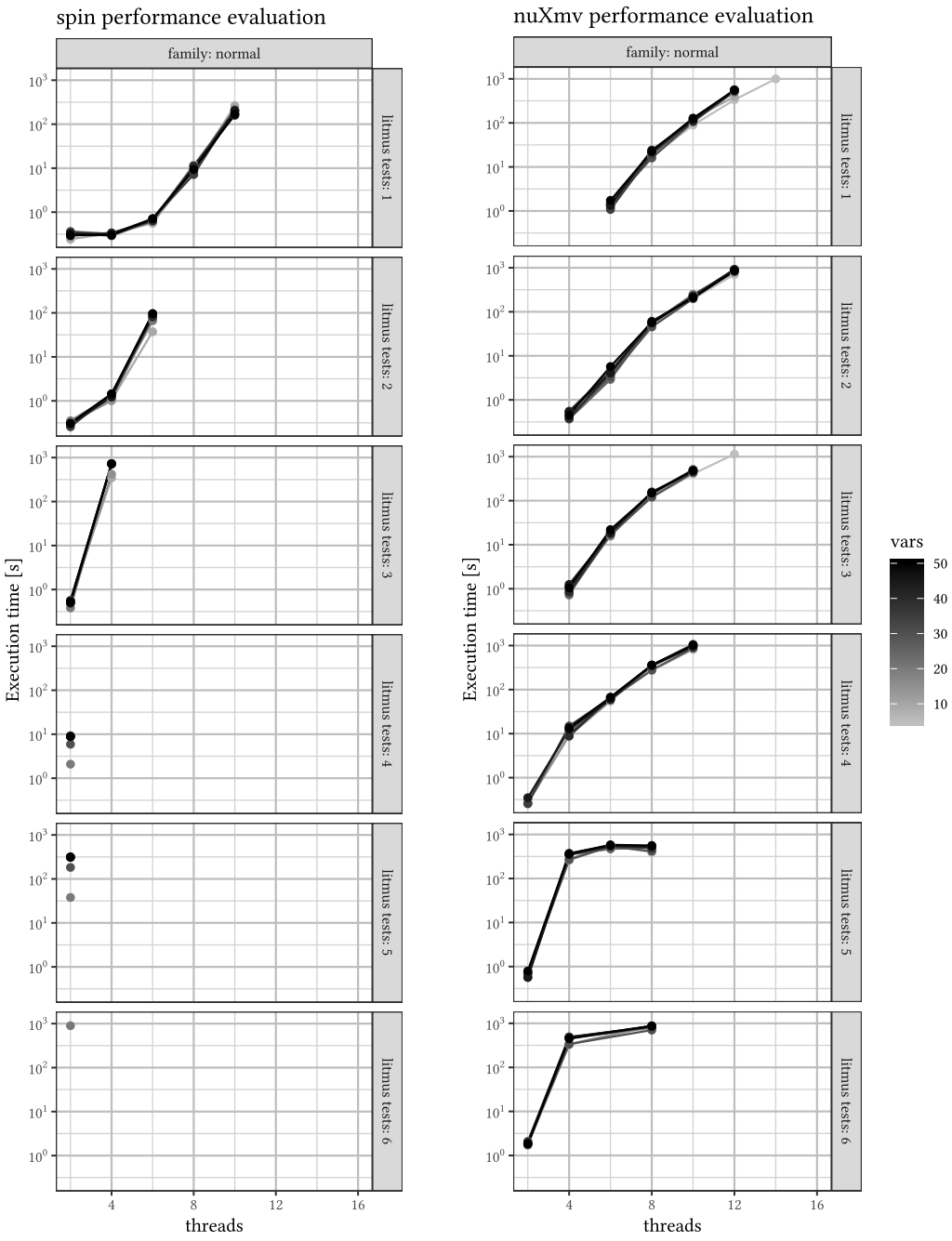

Fig. 12. Spin and nuXmv performance graphs, scaling by number of litmus tests, variables and threads. nuXmv could finish more tests in the given timeframe (1200s) and memory constraint (16GB). Furthermore, nuXmv scaled much better resulting in an almost linear regression, albeit on a logarithmic scale.

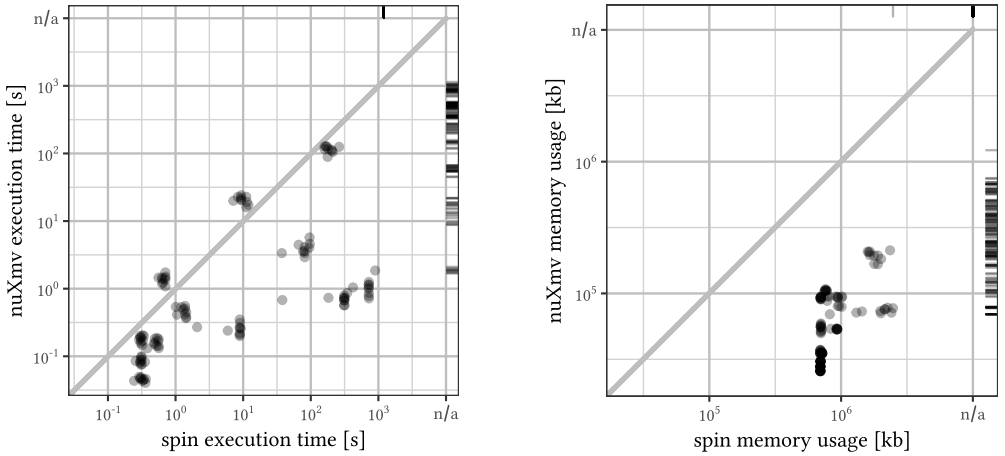

Fig. 13. Spin and nuXmv performance comparison. nuXmv performed better than spin both in terms of execution time and memory usage.

9 CONCLUSION AND FUTURE WORK

In this paper, we have proposed a new kind of formal verification problem to decide if a concurrent program will activate a fault in the memory consistency model of a processor when executed on it. We have outlined an exact solution based on the most recent advances of software and memory consistency model verification, and considered some approximative solutions. We have also implemented and evaluated a partial solution that is based on traditional model checkers.

The results indicate that the proposed problem can be solved, but traditional model checking tools are currently not suitable to address the full problem and they are also not prepared for this use case. Recent tools, however, are promising, and the gap between hardware and software verification is closing. We hope to provide a motivation for the integration and improvement of these methodologies and also a tool to system engineers to address the faults of existing hardware which would be expensive to repair or recall.

As we consider this only a first step in a new direction, we have a lot of future work. We plan to experiment with more approximative solutions and more model checkers, identifying the most efficient techniques. Furthermore, we plan to design purpose-built algorithms based on the methodology of [19] and [28] that can precisely solve the problem and work on new abstraction-based techniques to increase efficiency.

ACKNOWLEDGMENTS

The research reported in this paper was partially supported by the BME – Artificial Intelligence FIKP grant of EMMI (BME FIKP-MI/SC) and the Nemzeti Tehetség Program, Nemzet Fiatal Tehetségeiért Ösztöndíj 2018 (NTP-NFTÖ-18).

REFERENCES

[1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. 2018. Optimal stateless model checking under the release-acquire semantics. *PACMPL* 2, OOPSLA (2018), 135:1–135:29. https://doi.org/10.1145/3276505

[2] Jade Alglave. 2012. A Formal Hierarchy of Weak Memory Models. *Form. Methods Syst. Des.* 41, 2 (Oct. 2012), 178–210. https://doi.org/10.1007/s10703-012-0161-5

[3] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding cats: modelling, simulation, testing, and data-mining for weak memory. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. ACM, 40. https://doi.org/10.1145/2594291.2594347

[4] ARM Limited. 2011. *Cortex–A9 MPCore™ programmer advice notice: Read-after-read hazards (arm reference 761319)* (1 ed.). ARM Limited. http://infocenter.arm.com/help/topic/com.arm.doc.uan0004a/UAN0004A_a9_read_read.pdf

[5] ARM Limited 2011. *Cortex™–A9 MPCore®: Technical reference manual* (7 ed.). ARM Limited. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0407g/DDI0407G_cortex_a9_mpcore_r3p0_trm.pdf

[6] James Bornholt and Emina Torlak. 2017. Synthesizing memory models from framework sketches and Litmus tests. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. ACM, 467–481. https://doi.org/10.1145/3062341.3062353

[7] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. 1992. Symbolic Model Checking: 10^20 States and Beyond. *Inf. Comput.* 98, 2 (1992), 142–170. https://doi.org/10.1016/0890-5401(92)90017-A

[8] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. 2014. The nuXmv Symbolic Model Checker. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. Springer, 334–342. https://doi.org/10.1007/978-3-319-08867-9_22

[9] Soham Chakraborty and Viktor Vafeiadis. 2019. Grounding thin-air reads with event structures. *PACMPL* 3, POPL (2019), 70:1–70:28. https://doi.org/10.1145/3290383

[10] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. 2001. *Model checking*. MIT Press. http://books.google.de/books?id=Nmc4wEaLXFEC

[11] Michel Dubois, Christoph Scheurich, and Faye A. Briggs. 1986. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th Annual Symposium on Computer Architecture, Tokyo, Japan, June 1986*. IEEE Computer Society, 434–442. https://dl.acm.org/citation.cfm?id=17406

[12] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*. ACM, 110–121. https://doi.org/10.1145/1040305.1040315

[13] Kourosh Gharachorloo. 1995. *Memory Consistency Models for Shared-Memory Multiprocessors*. Technical Report. Stanford, CA, USA.

[14] Micha Hofri. 1990. Proof of a Mutual Exclusion Algorithm — a Classic Example. *SIGOPS Oper. Syst. Rev.* 24, 1 (Jan. 1990), 18–22. https://doi.org/10.1145/90994.91002

[15] Gerard J. Holzmann. 2004. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley.

[16] Intel Corporation 2015. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation.

[17] ISO/IEC 2010. *Programming languages — C - Committee Draft*. ISO/IEC. ISO/IEC 9899:201x.

[18] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *meltdownattack.com* (2018). https://spectreattack.com/spectre.pdf

[19] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2018. Effective stateless model checking for C/C++ concurrency. *PACMPL* 2, POPL (2018), 17:1–17:32. https://doi.org/10.1145/3158105

[20] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. ACM, 618–632. https://doi.org/10.1145/3062341.3062352

[21] Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2014. Pipe Check: Specifying and Verifying Microarchitectural Enforcement of Memory Consistency Models. In *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom, December 13-17, 2014*. IEEE Computer Society, 635–646. https://doi.org/10.1109/MICRO.2014.38

[22] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhattacharjee. 2016. COATCheck: Verifying Memory Ordering at the Hardware-OS Interface. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*. ACM, 233–247. https://doi.org/10.1145/2872362.2872399

[23] Daniel Lustig, Andrew Wright, Alexandros Papakonstantinou, and Olivier Giroux. 2017. Automated Synthesis of Comprehensive Memory Model Litmus Test Suites. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*. ACM, 661–675. https://doi.org/10.1145/3037697.3037723

[24] Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2015. CCICheck: using μhb graphs to verify the coherence-consistency interface. In *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*. ACM, 26–37. https://doi.org/10.1145/2830772.2830782

[25] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the gap between programming languages and hardware weak memory models. *PACMPL* 3, POPL (2019), 69:1–69:31. https://doi.org/10.1145/3290382

[26] Azalea Raad, Marko Doko, Lovro Rozic, Ori Lahav, and Viktor Vafeiadis. 2019. On library correctness under weak memory consistency: specifying and verifying concurrent libraries under declarative consistency models. *PACMPL* 3, POPL (2019), 68:1–68:31. https://doi.org/10.1145/3290381

[27] Pradeep S. Sindhu, Jean-Marc Frailong, and Michel Cekleov. 1992. Formal Specification of Memory Models. In *Scalable Shared Memory Multiprocessors*, Michel Dubois and Shreekant Thakkar (Eds.). Springer US, Boston, MA, 25–41. https://doi.org/10.1007/978-1-4615-3604-8_2

[28] Caroline Trippel, Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2017. TriCheck: Memory Model Verification at the Trisection of Software, Hardware, and ISA. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*. ACM, 119–133. https://doi.org/10.1145/3037697.3037719