Simulating Operational Memory Models Using Off-the-Shelf Program Analysis Tools

Dan Iorga, John Wickerson, *Senior Member, IEEE*, and Alastair F. Donaldson, *Member, IEEE*

***Abstract*—Memory models allow reasoning about the correctness of multithreaded programs. Constructing and using such models is facilitated by *simulators* that reveal which behaviours of a given program are allowed. While extensive work has been done on simulating *axiomatic* memory models, there has been less work on simulation of *operational* models. Operational models are often considered more intuitive than axiomatic models, but are challenging to simulate due to the vast number of paths through the model's transition system. Observing that a similar path-explosion problem is tackled by program analysis tools, we investigate the idea of reducing the decision problem of "whether a given memory model allows a given behaviour" to the decision problem of "whether a given C program is safe", which can be handled by a variety of off-the-shelf tools. We report on our experience using multiple program analysis tools for C for this purpose—a model checker (CBMC), a symbolic execution tool (KLEE), and three coverage-guided fuzzers (libFuzzer, Centipede and AFL++)—presenting two case-studies. First, we evaluate the performance and scalability of these tools in the context of the x86 memory model, showing that fuzzers offer performance competitive with that of RMEM, a state-of-the-art bespoke memory model simulator. Second, we study a more complex, recently developed memory model for hybrid CPU/FPGA devices for which no bespoke simulator is available. We highlight how different encoding strategies can aid the various tools and show how our approach allows us to simulate the CPU/FPGA model twice as deeply as in prior work, leading to us finding and fixing several infidelities in the model. We also experimented with applying three analysis tools that won the "falsification" category in the 2023 Annual Software Verification Competition (SV-COMP). We found that these tools do not scale to our use cases, motivating us to submit example C programs arising from our work for inclusion in the set of SV-COMP benchmarks, so that they can serve as challenge examples.**

***Index Terms*—Operational semantics, model checking, fuzzing, symbolic execution.**

I. Introduction

With the slowdown of Moore's law [1], systems are shifting towards more complex and heterogeneous architectures. In these heterogeneous systems, most computational elements can concurrently access shared memory. Therefore, to fully unlock all this computational power, software engineers need to build concurrent software that can correctly coordinate access to shared memory by diverse components. Achieving such correct coordination requires an accurate understanding of the system.

Memory models can aid in reasoning about these complex systems since they can be used to explore guarantees regarding the systems' behaviours. However, developing accurate memory models for complex systems is challenging.

Memory models are specified according to two main paradigms: *operational* and *axiomatic*. An operational model is an abstract representation of the actual machine, described by *states* that represent idealised components such as buffers and queues, and the legal *transitions* between these states. On multicore systems, there may be several available transitions from any given state, and hence an exponential blow-up in the number of paths to explore. On the other hand, an axiomatic model defines relations between memory accesses to constrain the allowed and disallowed behaviours. Simulation of axiomatic models can be orders of magnitude faster than the simulation of operational models [2], [3], but operational models are often considered more intuitive and there is a great deal of recent work that relies on operational models [4], [5]. The slower simulation time associated with operational models, together with the demand for operational simulators, motivates us to focus on them, to investigate ways to reduce the engineering effort required to obtain them in the first place, and to make simulation more efficient.

An operational memory model takes as input a sequence of instructions for each concurrent component of the system and a description of a final state of interest. This input is usually called a *litmus test* [6]. It then facilitates searching for transitions of the system that might lead from an initial state to the final state of interest. A trace that leads to the final state indicates that this behaviour is *allowed*; if no such trace exists, it is *disallowed*. Non-determinism arises due to the order in which the concurrent components issue the instructions and due to the internals of the memory system (such as flushing policies for buffers and caches). Once a trace that reaches a state of interest has been found, the programmer or memory-model engineer can use the simulator to step through the trace in detail to understand its behaviour better.

Manuscript received 15 October 2022; revised 21 July 2023; accepted 7 October 2023. Date of publication 24 October 2023; date of current version 12 December 2023. This work was supported in part by the EPSRC via the IRIS Programme under Grant EP/R006865/1 and in part by the HiPEDS Doctoral Training Centre under Grant EP/L016796/1. Recommended for acceptance by W. Visser. *(Corresponding author: Dan Iorga.)*

Dan Iorga and Alastair F. Donaldson are with the Department of Computing, Imperial College London, SW7 2AZ London, U.K. (e-mail: d.iorga17@imperial.ac.uk; alastair.donaldson@imperial.ac.uk).

John Wickerson is with the Department of Electrical and Electronic Engineering, Imperial College London, SW7 2AZ London, U.K. (e-mail: j.wickerson@imperial.ac.uk).

Digital Object Identifier 10.1109/TSE.2023.3326056

© 2023 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

Operational memory models are principally intended for use by engineers who need to understand specific details of how a memory model behaves. They thus need to scale to reasonably-sized litmus tests. However, they are not intended for analysis of full-blown software applications. The scope of our work is thus intentionally limited to the scenario of litmus test analysis.

A state-of-the-art simulator for operational memory models is RMEM [7], which has been used to simulate the memory models of ARM [3], [8], Power [9], [10], [11], RISC-V [5] and x86 [12]. Building a *bespoke* simulator such as RMEM requires a lot of engineering effort: not only must the memory model of interest be encoded, but algorithms for efficient reachability analysis must be implemented. For example, the RMEM simulator is still actively being updated four years after its initial release and already contains more than 60k lines of OCaml code.

Reachability has been studied extensively in the context of program analysis, and a range of off-the-shelf tools that attempt to decide whether a program can reach a particular state are available for several languages. This leads to the following idea: instead of implementing a bespoke memory model simulator, why not implement the logic of the memory model as a program that takes a particular test scenario as input? Determining whether the test scenario is allowed would then boil down to determining whether a particular state of the program that encodes the memory model is reachable when executed on an input describing the scenario of interest, and off-the-shelf reachability analysis tools for the language of interest could be leveraged to answer this question. Subsequent detailed examination of traces would then be possible by stepping through the simulator code using a standard debugger.

As discussed above, an operational memory model only needs to capture the concurrency semantics of individual instructions, because litmus tests typically comprise a number of straight-line instruction sequences. This means our approach does not require modelling a complete interpreter for an instruction set architecture or programming language (which would be needed in order to perform analysis of complete software applications, and is not the aim of our work).

We report on our experience putting this idea into practice, using C as the implementation language. We focus on C not out of any particular fondness for the language, but due to the availability of a diverse range of C analysis tools. While previous work has already created executable memory models by modifying tools such as Java PathFinder [13] or XSB [14], the advantage of our approach is that we do not require any modification to the tools and can easily plug in different program analysis tools and take advantage of the strengths of each one.

Our work is divided into two parts: (1) a study of the strengths and weaknesses off three off-the-shelf C analysis tools and a comparison with RMEM in the context of the well-known x86 memory model, and (2) an in-depth study of a much more recent CPU/FPGA memory model, enabled by our use of off-the-shelf tools.

A. First Part: x86 Case Study

In the first part of this article, we are interested in the following top-level research questions:

RQ1 Can reducing the problem of memory model simulation to the analysis of a C program yield performance competitive with bespoke simulators?

RQ2 Of the variety of C analysis tools that we consider, which are most effective for the memory model simulation approach that we propose?

We use the x86 memory model since it is simple and widely-used. We investigate our idea of reducing to C and then leveraging existing tools with respect to a number of diverse analysis tools for C: a SAT-based model checker, CBMC [15]; a dynamic symbolic execution engine, KLEE [16], and three coverage-guided fuzzers, libFuzzer [17], Centipede [18] and AFL++ [19]. Of these, CBMC is a fully symbolic analyser, the coverage-guided fuzzers are fully dynamic analysers, and KLEE mixes symbolic and dynamic analysis.

An important property these tools have in common, which makes them suitable for memory model simulation, is that they are *precise* with respect to bug finding: they do not use abstraction (as is employed by many static analysis tools), and therefore they can identify inputs that are *guaranteed* to reach particular program locations. A difference between the tools is the extent to which they can perform exhaustive search of the state space of a particular simulation scenario. Being based on randomisation, the fuzzers cannot provide such guarantees. However, because both KLEE and CBMC are systematic, they can in principle perform exhaustive exploration of a simulation scenario, provided that the C encoding of the memory model avoids infinite paths.

The main finding from this first study is that coverage-guided fuzzing is extremely effective at confirming allowed behaviours for the litmus test configurations we consider, generally vastly outperforming both model checking and symbolic execution. This further supports the idea of using coverage-guided fuzzing for search tasks by encoding such tasks as bug-finding problems; e.g., coverage-guided fuzzing was used successfully as a means of showing satisfiability of floating-point SMT formulas [20], by transforming a formula to a program that takes a valuation of the free variables of a formula as input, and contains an error location that is reachable if and only if the valuation is a satisfying assignment.

Furthermore, we find that the *coverage-guided* feature that libFuzzer, Centipede and AFL offer is essential—we also present results using a naïve fuzzer that is not guided by coverage, which does not perform nearly as well as the coverage-guided ones.

While specific optimisations can enable RMEM to scale better than KLEE and CBMC, it is still outperformed by libFuzzer and Centipede for larger thread counts. Furthermore, the out-of-the-box tools do not require any extra engineering effort.

Despite the advantages of coverage-guided fuzzing, for particularly complex litmus tests where an allowed behaviour is exhibited by only a tiny fraction of paths, SAT-based model checking is able to demonstrate that the behaviour is allowed while exploration using coverage-guided fuzzing gets lost. Furthermore, because symbolic execution and SAT-based model checking are capable of *exhaustive* exploration—unlike fuzzing—they can be used to demonstrate that certain memory model behaviours are *disallowed*.

B. Second Part: CPU/FPGA Case Study

We then apply our ideas to a memory model for which no bespoke simulator exists: the X+F memory model that we recently formalised [21]. This is the memory model of a system that combines an Intel Xeon CPU with a field-programmable gate array (FPGA). It poses challenges due to the complexity of shared-memory interactions between CPU cores and FPGA logic. Our original work gave a C encoding of this memory model and used CBMC to validate it against a supposedly-equivalent axiomatic memory model, but the evaluation was restricted to small litmus tests due to scalability limitations of CBMC. Given our finding from Part 1, that coverage-guided fuzzing tools perform well for x86 litmus tests and that coverage is an aiding factor, we ask the following research questions in the context of the X+F model:

RQ3 How does the manner in which the memory model and litmus test are encoded as a C program impact the performance of the different tools?

RQ4 Can our approach allow more in-depth analysis of the X+F memory model, allowing it to be better validated against its axiomatic counterpart?

Thanks to the better scalability of coverage-guided fuzzing compared with SAT-based model checking, we were able to perform a substantially deeper analysis than in prior work. This allowed us to find four infidelities in the X+F axiomatic memory model. We have updated the model so that it now accounts for additional ordering guarantees that had been overlooked in our prior work.

In summary, our main contributions are:

- the idea of reducing the decision problem of "whether a given operational model allows a given program behaviour" to the decision problem of "whether a given program is safe", and leveraging off-the-shelf tools to answer this question,
- a report on our experience using a number of off-the-shelf C program analysis tools to facilitate memory model simulation, comparing them to a bespoke memory model simulator,
- a case study leveraging the scalability of coverage-guided fuzzing to allow deeper analysis of a hybrid CPU/FPGA memory model, leading to the finding and fixing of four infidelities in an axiomatic description of this memory model, and
- insights into how a program encoding of a memory model can be made to play to the strengths of a coverage-guided fuzzing tool.

Our experience suggests that the manual effort associated with writing a bespoke memory model simulator might be better directed into encoding the memory model as a program in a mainstream language for which an array of analysis tools is available. This allows these tools to be leveraged for memory model analysis, avoiding the need to implement state-space exploration algorithms.

This article is structured as follows. In Section II we introduce memory models and give an overview of our approach. Section III addresses research questions **RQ1** and **RQ2**. Section IV addresses **RQ3** and **RQ4**. We discuss related work in Section V and conclude in Section VI.

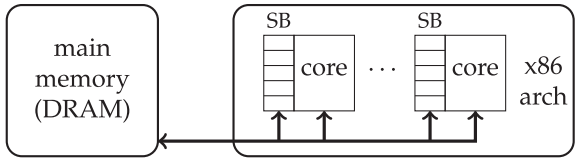

Fig. 1. A pictorial representation of the x86 memory model. The store buffers (SB) present in each core can cause weak memory behaviours. Each core will first write to its store buffer before committing to main memory.

II. Overview of Our Approach

In this section, we provide background on weak memory models and litmus tests (Section II-A), show how the x86 memory model can be encoded as a C program (Section II-B), discuss the analysis tools considered in the article (Section II-C), discuss different options for encoding litmus tests (Section II-D) and illustrate how program analysis tools can be used to simulate memory models (Section II-E).

A. Background on Memory Models

Modern architectures do not implement sequential consistency (SC) as defined by Lamport [22]. As a result, programmers cannot expect their programs to access memory in the order in which loads and stores appear in their source code, without additional synchronisation. Accessing main memory has high latency, and optimisations are required to hide this latency. Optimisations that cause **weak memory** effects are present in all modern architectures. These weak memory effects are only observable when multiple threads access the same data in shared memory. The possible reorderings that can result from this are often quite subtle.

A **memory model** defines the memory-related behaviours a system permits. As an example, let us consider the operational memory model of the x86 architecture [12], illustrated in Fig. 1. Each core has its own *store buffer*. When a core issues a write to memory, the write temporarily resides in the store buffer, and all writes in a store buffer are bulk-transferred to main memory periodically. When loading from a location, a core will first check whether a write to that location is pending in its store buffer. If so, it will return the value associated with that write, thus avoiding the expensive operation of reading from main memory. As a result, each core has a different view of the system's memory: a core may be able to observe the writes that it has issued before they can be observed by other cores. These different views of main memory can lead to unintuitive behaviours, where cores observe memory operations as having occurred in an order that is not *sequentially consistent*: it does not correspond to any interleaving of instructions executed by individual threads. Programmers can recover sequential consistency with the aid of special *fence* operations, which force store buffers to be flushed so that writes become visible to all cores. Because fence operations are expensive, they should be used sparingly.

Litmus tests are small concurrent programs designed to reveal whether a specific memory model behaviour can occur. A litmus test usually comprises a sequence of shared memory write and read operations, followed by an assertion over the

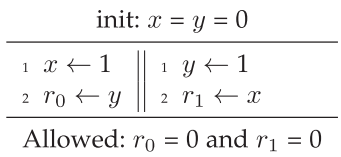

Fig. 2. Store buffering using two threads.

Listing 1. The structure of an operation

```
struct Operation {
  Opcode type;
  int var;
  int val;
};
```

Listing 2. The possible actions

```
enum Action {
  CPU_THREAD,
  FLUSH_BUFFER
};
```

Listing 3. The pseudocode of the mechanised x86 memory model.

```
int sim_steps = choose(SIMULATION_STEPS);
for (int i = 0; i < sim_steps; i++) {
  // Can be one of the n threads in the system
  int thread = choose(NUM_THREADS);
  // Can be either CPU_THREAD or FLUSH_BUFFER
  Action action = choose(NUM_ACTIONS);
  switch (action) {
    case CPU_THREAD:
      if (!thread_ops[thread].empty()) {
        Operation op = thread_ops[thread].pop();
        if (op.type == WRITE) {
          write_to_buffer(thread, op.var, op.val);
        }
        if (op.type == READ) {
          read_buffer_or_memory(thread, op.var);
        }
        break;
      }
    case FLUSH_BUFFER:
      if (!buffer[thread].empty()) {
        flush_buffer(thread);
        break;
      }
  }
}
check_litmus_test();
```

values that were observed by reads. A litmus test should be executed many times on a processor of interest to gain confidence as to whether or not the assertions associated with the litmus test hold, because concurrent systems with weak memory models are nondeterministic. Moreover, it is possible that some weak behaviours only manifest when the system is under heavy stress [23].

Specific litmus tests have been designed to characterise particular architectural features that might give rise to certain weak behaviours. For example, in Fig. 2 the well-known *store-buffering* litmus test is illustrated. These litmus tests can reveal the write buffers of the x86 memory model seen in Fig. 1. The test requires both writes to be buffered in order for the CPU cores to observe the old values of variables x and y. If this happens, the old values of the variables will be observed.

B. Reducing x86 Analysis to C Reachability

Our approach is to encode memory models using the C programming language, and to leverage off-the-shelf C program analysis tools for simulation. To make this idea concrete, in Listing 3 we sketch a C implementation of the x86 memory model. The intention is that this code handles all the complexities of the *concurrency* semantics for x86, but only supports very basic *sequential* semantics, handling simple sequences of loads, stores, and fence instructions, as are found in litmus tests. In particular, our approach does not require implementing a full-blown interpreter for an instruction set architecture.

The sequence of instructions describing the litmus test is initialised in the `thread_ops` queue of C structures and Listing 1 shows the structure of a thread operation. Whenever a nondeterministic value is required, we use the `choose` function. This is only a placeholder function and will be replaced by the corresponding API of a program analysis tool—a read from a source of random data when using a fuzzer, and an operation to generate a boolean decision on an unconstrained symbolic variable in a symbolic analysis tool. We first use this function at the beginning of the simulation, for the number of simulation steps since we do not know many steps are required for each litmus test. Afterwards, the main loop of the program should ensure that this number of simulation steps are executed. At each loop of the simulation, the `choose` function will be invoked again to select the `action` that will be performed by a `thread`. The possible actions that the system can make are declared in Listing 2.

Each case-statement corresponds to an operational semantic rule and is guarded by an if-statement that verifies if the preconditions of the rule holds. If the `CPU_THREAD` action is chosen, the simulator will first check if there are any thread operations left for that specific thread, remove the next one from the `thread_ops` queue and attempt to process it. If this is a read operation, the x86 simulator will search for that value in the thread buffer and if it does not find it there, will search for it in main memory. Correspondingly, if the operation is a write, the simulator will add this operation to the store buffer. If a `FLUSH_BUFFER` action is chosen and the buffer is not empty, the simulator will transfer the data from the buffer to main memory.

It is possible to commit to an action (via the switch) before realising that its guard does not actually hold. In such a case, nothing would happen on this iteration. At the end of the execution, we check whether the program has reached the state that the litmus test describes. By exploring all the possible combinations of `thread` and `action` allowed values, the simulation will explore all the possible outcomes, given the `thread_ops` provided as input.

Generalisation We can generalise the structure of an operational model with the following general structure:

1) *Define opcodes, operation structure and actions.* Different systems will have different operation types and will perform different types of actions.
2) *Initialise all memory model components.* In our example, this means having store buffers be initially empty. In a more complex memory model there might be more complex components such as pools or caches.
3) *Initialising a per-thread queue of instructions.* At this point, the simulator should interface with the per-test harness so that it can initialise the queue of instructions.

4) *Write the state machine as a loop.* At each simulation step, a decision will be made nondeterministically as to whether the machine will consume user input or if one of the components of the machine will perform a transition. If the component has multiple choices (such as being able to flush different elements), the choice between them will be made nondeterministically. (The manner in which nondeterminism is resolved in practice depends on the tool that is used for analysis, as discussed above.)
5) *Check if the assertion holds*. Here the simulator should again interface with the per-test harness so that it can check if the re-orderings described by it occurred.

We are unlikely to have access to all the microarchitectural features of the system. Therefore, we cannot know the exact size of all the buffers in the model. However, we can choose sizes for these buffers that are large enough to allow all reorderings that the system is capable of performing, but no larger, so as not to add any unnecessary burden on simulation. Even though this kind of simulation is limited by its bounded nature, we can empirically verify that the bounds are sufficiently large for our litmus tests. We do this by adding `assert` statements within our model that verify if the buffers ever reach their total capacity and limit the paths available through the program.

Having described this general recipe for extending the model, we show in Section IV how we can put it into practice for modelling a more intricate model for a combined CPU/FPGA system.

C. The Analysis Tools We Consider

Recall that our approach involves encoding the operational semantics of a memory model as a nondeterministic C program, where the nondeterminism accounts for thread interleavings, and reordering in the memory subsystem. This program, combined with an input describing the litmus test, is then suitable for (a) exhaustive simulation, where a tool attempts (dynamically or symbolically) to explore all possible paths through the program, or (b) randomised simulation, where a tool repeatedly explores paths, resolving nondeterminism via randomisation.

We now discuss the analysis tools that are considered later in the article—CBMC, KLEE (which allow for exhaustive simulation), and libFuzzer, Centipede and AFL++ (which allow for randomised simulation).

1) CBMC: CBMC [15] is a bounded model checker for C and C++ programs. Given an entry point function to the program under analysis, and a maximum unwinding depth d for program loops, CBMC constructs a logical formula asserting that there exists an input to the entry point function that will cause the program to fail with an error along at least one path, requiring no more than d iterations of any individual loop. This formula is discharged by a SAT or SMT solver (CBMC uses MiniSat [24] by default). If the formula is *satisfiable*, the satisfying assignment provides an input to the entry point that will cause the program to fail. If the formula is *unsatisfiable* this proves that there does not exist an input that can cause the program to fail within the bounds of the unwinding depth d; that is, the program is *correct* up to this bound, hence the name *bounded* model checking. However, this bounded guarantee says nothing about whether there exist inputs that trigger deeper bugs.

CBMC also supports *unwinding assertions*, which assert that no paths exceeding the loop bound d exist. If verification succeeds when unwinding assertions are enabled, this shows that (a) no bugs exist that can be revealed up to loop bound d, and (b) no paths exist that exceed loop bound d. Together, these constitute a proof of correctness.

Because CBMC is purely symbolic, yielding a single SAT or SMT formula encoding all executions of a program up to a certain depth, it can potentially scale poorly due to the formula getting large, leading to long SAT/SMT solving times. On the other hand, solving this big formula is all that needs to be done. Since the formula that CBMC constructs encodes *all* paths through the program up to the given depth, CBMC can be used to verify conclusively that a given scenario is *not* possible for execution traces below a certain length—or, when unwinding assertions are used—for all traces. We chose CBMC as our model checker because it is widely used, robust and practical and is well suited for system-level modelling.

2) KLEE: KLEE [16] is a dynamic symbolic execution engine for C and C++ programs built on top of the LLVM compiler infrastructure. Like CBMC, KLEE operates on an entry point function. The user can mark some of the data consumed by this function as *symbolic*. KLEE maintains a set of paths that are under exploration, maintaining for each path (a) the next program instruction to be executed on the path, (b) concrete values for all variable and memory locations that are not symbolic, and (c) a set of constraints, called a *path condition*, restricting the values of symbolic data.

The KLEE execution engine works by repeatedly selecting a path from the set and progressing the path by executing a number of instructions. When the path reaches a decision point—e.g. the guard of an `if` statement—KLEE uses an SMT solver to assess which outcomes of the decision are feasible. It does this by querying whether the path condition is feasible when extended with the boolean guard of the decision, or the negation of this guard. If only one side of the decision is feasible, KLEE appends the boolean expression associated with the outcome of the decision to the path's path condition. If *both* are feasible, KLEE *forks* the path so that in the future two different path extensions will be considered: one that explores the "then" side of the decision, with its path condition extended by the guard of the decision; the other that explores the "else" side of the decision, with path condition extended by the negation of the guard.

When an assertion is encountered, or another "dangerous" operation, such as division (which might be by zero) or an array access (which might be out of bounds), KLEE uses the SMT solver to query whether it is possible for the path condition to be satisfied *and* or the assertion or dangerous operation to fail. In cases where the SMT solver returns a satisfying assignment, KLEE mines the assignment to yield concrete values for the symbolic data consumed by the entry point function that will lead to the error occurring.

KLEE uses a number of heuristics to decide how to explore the set of paths that it maintains, and makes use of a

counter-example cache to aim to avoid issuing redundant SMT queries [16].

KLEE has the potential to scale better than CBMC since solving formulas on a per-path basis might be easier than solving a single large formula encoding all program paths. Like CBMC, KLEE can verify that a scenario is not feasible if the associated program has a finite number of paths (though the *path explosion* problem means this is not always feasible in practice).

3) libFuzzer, AFL++ and Centipede: These tools are all examples of *mutation-based*, *coverage-guided* fuzzers [25]. To use tools of this kind, the developer writes a *fuzz* target: a function that takes a sequence of bytes as input, and that should use the sequence of bytes to invoke a function of the system under test (SUT). For example, when fuzzing a compression library, the sequence of bytes might be fed to a "decompress" function as a candidate sequence of compressed bytes; when testing a compiler the sequence might be treated as a string and fed to the compiler front-end.

Mutation-based fuzzers work by starting with an initial set of inputs (which can either be existing test cases, or default, simple inputs such as a buffer of zeros or a randomly-initialised buffer), and repeating the process of (1) mutating an input at random (e.g. by deleting, inserting or shuffling some of the bytes), and (2) feeding the mutated input to the fuzz target. The hope is that mutated inputs may exercise the SUT in new and potentially unexpected ways that might trigger bugs. Mutation-based fuzzing focuses on triggering *crash* bugs, arising due to violation of user-written assertions, or erroneous use of programming language features (e.g. buffer overflows or division by zero in C).

Coverage-guided mutation-based fuzzers (henceforth referred to simply as coverage-guided fuzzers) expand this idea as follows. Before fuzzing commences, the SUT is compiled in a manner that enables efficient *coverage instrumentation*, so that the fuzzer can remember which statements and branches of the SUT have already been exercised by previous inputs. During fuzzing, the fuzzer maintains a corpus of "interesting" inputs that should be considered for mutation. Initially this is just the set of starting inputs described above. Fuzzing then involves repeating the process of (1) selecting an input from the corpus, (2) randomly mutating the input, (3) feeding the mutated input to the fuzz target, and (4) if the input achieves new code coverage, adding it to the corpus of interesting inputs so that it is considered for further mutation in the future. This is a simple example of an evolutionary algorithm [26]: mutation is used to evolve a corpus of inputs that exercise the SUT increasingly thoroughly, with code coverage serving as the fitness function. The hypothesis behind coverage-guided mutation-based fuzzing is that inputs that cover new code have a higher chance of detecting new crash bugs compared with inputs that merely exercise already-covered code.

Due to the need to instrument the SUT with coverage information, coverage-guided fuzzing is often called *grey-box* fuzzing [25].

AFL++ [27], based on the ground-breaking AFL fuzzer [19] is an *out-of-process* coverage-guided fuzzer. The SUT must be compiled in a special manner such that coverage of edges in the control flow graph of the compiled code can be efficiently tracked. The AFL++ driver then manages the coverage-guided mutation-based fuzzing process, launching the SUT in a fresh process for each test input; the use of separate processes is why these fuzzers are termed "out-of-process". AFL++ has been used to find numerous security-critical defects [28]. We experiment with AFL++ rather than its predecessor, AFL, because AFL++ aims to be "a superior fork to Google's AFL—more speed, more and better mutations, more and better instrumentation" [29], and unlike the original tool it is still actively developed. The tools support a set of default, domain-agnostic mutators that perform byte-level manipulation. They also provide support for user-supplied "custom mutators", which we do not consider in this work.

In contrast, libFuzzer [17] is an *in-process* coverage-guided fuzzer: the fuzz driver is compiled and linked with the SUT, so that the fuzzer and SUT run in the same process. This leads to improved efficiency since it avoids the overhead of continually launching processes. The downsides are that: when a crash occurs, the entire fuzzer crashes, so that it is necessary to invoke libFuzzer from a wrapper script that will continually re-launch it upon a crash; global state initialised at program load time will not be automatically reset when the fuzzer tries successive inputs. The tool is embedded into the Clang/LLVM compiler framework, performs code coverage and mutation in a similar manner to AFL++, and provides a similar set of built-in mutators as well as support for custom mutators. libFuzzer is the main fuzzing engine used by Google's ClusterFuzz project [30], which has been used to find tens of thousands of bugs in large software projects such as the Chromium web browser. It is also deployed in OSS-Fuzz [31], a deployment of the ClusterFuzz technology targeting open source projects.

Centipede [18] is a work-in-progress, out-of-process successor of libFuzzer, designed to allow highly parallel fuzzing that can be distributed across clusters of machines. This is where the main innovations in Centipede lie; to our knowledge the core fuzzing engine is based directly on libFuzzer. For completeness, throughout the paper we present results for Centipede in addition to libFuzzer, since libFuzzer is now in maintenance mode.

Inter-operability between fuzzers The libFuzzer and Centipede tools are directly inter-operable: Centipede advertises "Out-of-the-box support for libFuzzer-based fuzz targets" [18]. By default, AFL++ offers a different interface for writing fuzz targets compared with libFuzzer and Centipede, but a simple adapter is available that allows AFL++ to be used with libFuzzer-style fuzz targets [32]. As a result, once a libFuzzer-style fuzz target has been written for an SUT, the SUT can be tested using Centipede, AFL++ with either no or very minimal effort.

Use of coverage-guided fuzzing in our work Coverage-guided fuzzing is a completely dynamic technique, and although coverage instrumentation records which statements and branches have been exercised, fuzzing tools cannot perform systematic exploration: they neither remember the complete set of inputs that have been tried, nor the precise paths explored by inputs that are committed to the corpus. As a result,

Listing 4. Encoding the litmus test to favour high coverage.

```
1  check_litmus_test(){
2    // Check if final state reached
3    if (writesExecuted < totalWrites) return 0;
4    if (readsExecuted < totalReads) return 0;
5
6    // Check if the expected order was reached
7    if (timeEvent[0] < timeEvent[1]) return 0;
8    ...
9    if (timeEvent[n-1] < timeEvent[n]) return 0;
10
11   // Assert if register values are as expected
12   assert(reg[0]==a && ... && reg[m]==z);
13 }
```

Listing 5. Encoding the litmus test for fewer paths

```
1  check_litmus_test(){
2    // Check if final state reached
3    int reached = 1;
4    reached &= (writesExecuted == totalWrites);
5    reached &= (readsExecuted == totalReads);
6
7    // Check if the expected order was reached
8    int observed = 1;
9    observed &= (timeEvent[0] < timeEvent[1]);
10   ...
11   observed &= (timeEvent[n-1] < timeEvent[n]);
12
13   // Assert if register values are as expected
14   assert(reached & observed &
15           reg[0]==a & ... & reg[m]==z);
16 }
```

they can be used to find bugs but not to prove absence of bugs. In the context of our work, this means that they have the potential to show that a particular memory model-related behaviour is *allowed* (by finding a witness), but they cannot show that a behaviour is *disallowed* (which requires exhaustive search).

Compared with CBMC and KLEE, being purely dynamic coverage-guided fuzzing does not require potentially expensive SAT queries, and can quickly execute the program with many different inputs and monitor for potential errors, which has the potential to rapidly yield counter-examples.

D. Designing the Per-Test Harness

Reasoning about a litmus test involves combining the model with a test-specific harness. Each test-specific-harness will describe the sequence of operations for each concurrent component of the system, check if the simulation has run for a sufficient number of steps and if the reordering has been reached and finally, assert whether a particular outcome is observable.

The choices of encoding the litmus test may affect the scalability of the program analysis tool that is subsequently applied to the resulting program. An essential dimension for consideration here is branching: a piece of code that exhibits a lot of branching can be well-suited for coverage-guided fuzzing. This is because the branching will lead to various coverage targets, which can help identify diverse program inputs. Previous work [33] has already shown the effectiveness of increased branching for coverage-guided fuzzing.

In contrast, additional branching may lead to path explosion for symbolic tools if the branches rely on symbolic conditions. We have experimented with adding "early exit" branches in our post-condition checking in a manner that provides extra opportunities for coverage-guided feedback for the fuzzers but does not hinder KLEE. By construction, at the late stage of execution where these branches occur, KLEE will have already resolved all nondeterminism associated with simulating the litmus test of interest. Thus, while resolving this nondeterminism will have led to KLEE forking many different paths, on a per-path basis the additional branches occurring in the post-condition checks resolve deterministically and do not contribute to further path explosion.

Listing 4 shows an encoding designed to exhibit more branching, while Listing 5 shows an encoding designed to exhibit less branching. These encodings are merely two syntactically different methods for representing the same underlying program semantics. The engineering effort required to switch between them was minimal—we did not measure it, but estimate that it was no more than an hour of work, including the time taken to carefully check the changes that were made.

In both cases, a set of preconditions must ensure that the program has executed enough simulation steps and correctly models the test. Lines 2–4 in Listing 4 and lines 3–5 in Listing 5 check if the executed writes and reads (described by the `writesExecuted` and `readsExecuted` variables) are equal to the writes and reads in the litmus test (described by the `totalWrites` and `totalReads` variables). More complex models might require additional checks, such as verifying that buffers are empty. Furthermore, lines 6–9 in Listing 4 and lines 7–11 in Listing 5 ensure that the events in the litmus test, (recorded in the `timeEvent` array) have occurred in the expected order. The first encoding will immediately return when one of these preconditions is not met, while the second encoding will set the *reached* and *observed* variables to record the status of all preconditions.

Whenever a read operation is performed, its results are stored in the `reg` array. Assertions over this array can then be used to check whether the values observed by loads correspond to the values expected by the litmus test. While Listing 4 will simply assert on the values observed in the registers, Listing 5 will only assert if the *reached* and *observed* variables have been set accordingly. Recall that the short-circuiting `&&` operation will only evaluate its second operand if the first one is true, while the `&` operator will evaluate all terms regardless. As a result, the use of `&&` in Listing 4 leads to additional branching points and thus more code to be potentially covered, while the use of `&` in Listing 5 leads to fewer branching points and less code to be potentially covered.

We expect some tools to favour different encoding types and explore these options in Section IV-C by automatically generating litmus tests using the two alternative options.

E. Using C Analysis Tools to Simulate Memory Models

The *choose* function from Listing 3 shows all the points in the C program where non-determinism needs to be explored by the program analyser. To allow this, we need to furnish the analyser with a means of exploring this non-determinism. Here,

Listing 6. The CBMC adaptation of the x86 memory model

```
// Non-deterministic number of simulation steps
int sim_steps = nondet_int();
for (int i = 0; i < sim_steps; i++) {
  // Non-deterministic thread choice
  int thread = nondet_thread(NUM_THREADS);
  // Non-deterministic action choice
  Action action = nondet_action(NUM_ACTIONS);
  switch (action) {
    case CPU_THREAD:
      assume(!thread_ops[thread].empty());
      Operation op = thread_ops[thread].pop();
      if (op.type == WRITE) {
        write_to_buffer(thread, op.var, op.val);
      }
      if (op.type == READ) {
        read_buffer_or_memory(thread, op.var);
      }
      break;
    case FLUSH_BUFFER:
      assume(!buffer[thread].empty());
      flush_buffer(thread);
      break;
  }
}
check_litmus_test();
```

Listing 7. The KLEE adaptation of the x86 memory model

```
// Make number of simulation steps symbolic
klee_make_symbolic(sim_steps, sizeof(sim_steps));
for (int i = 0; i < sim_steps; i++) {
  // Make current thread symbolic
  klee_make_symbolic(thread, sizeof(thread));
  // Make current action symbolic
  klee_make_symbolic(action, sizeof(action));
  switch (action) {
    case CPU_THREAD:
      if (thread_ops[thread].empty())
        klee_silent_exit();
      Operation op = thread_ops[thread].pop();
      if (op.type == WRITE) {
        write_to_buffer(thread, op.var, op.val);
      }
      if (op.type == READ) {
        read_buffer_or_memory(thread, op.var);
      }
      break;
    case FLUSH_BUFFER:
      if (buffer[thread].empty())
        klee_silent_exit();
      flush_buffer(thread);
      break;
  }
}
check_litmus_test();
```

we describe the peculiarities of each approach and how the model needs to adjust for each specific tool.

Listings 6 to 8 are instantiations of the pseudocode of Listing 3 showing how the nondeterminism is modelled using CBMC, KLEE and the fuzzer tools, respectively. Our practical implementation uses preprocessor macros to allow these code variants to co-exist as one piece of source code, for ease of maintenance. Adapting the code to each tool required us to understand the features offered by the tools and to have some knowledge of how the tools work. However, the code adaptations themselves were very straightforward; we did not measure the engineering time required, but we estimate that it was less than three hours in total.

CBMC-based Validation CBMC will symbolically unwind the main simulation loop up to a certain *loop-unwind depth*, which is given as a parameter to the program. Since the litmus test will be encoded with a finite amount of operations and the model will not include "no-op" transitions, we know that the model will eventually reach its final state where no more transitions can be taken. However, the final state may never be reached if the unwind depth is not sufficiently high enough. We can place *assert* statements to empirically verify if certain intermediate states have been reached. Furthermore, we invoke CBMC with the `--unwinding-assertions` option, whereby it checks that unwinding the program further does not lead to any more states being explored. In this mode, CBMC can *prove* that the program under test is free from assertion failures: if an insufficiently large unwinding depth for loops is used then an "unwinding assertion" fails, indicating that a higher bound is required for the proof to succeed. This only works because we have modelled our program to contain a final state and thus have avoided infinite loops. When using CBMC for memory model analysis, we use per-litmus test information to estimate a suitable unwinding depth, and use a script to iteratively increase this until it is deep enough.

The operational semantic rule triggered at each point in the simulation is chosen non-deterministically. The premises of the operational rules are implemented using `assume` statements. The query that CBMC sends to the SAT/SMT solver includes constraints ensuring that any paths on which the guards of `assume` statements are not met are deemed infeasible. This ensures that the premises of a rule must be met for the rule to trigger.

CBMC for x86: Recall the code snippet from Listing 3 and note the updates in Listing 6 required for CBMC. The CBMC version of this model utilises a `nondet_int()` statement to allow the simulation to execute a nondeterministic number of steps. Furthermore, `thread` and `action` are similarly chosen non-deterministically by corresponding statements. To guarantee that the premises of semantic rules are satisfied, `assume` statements are used. The final `assert` statement verifies if the reordering has been detected.

KLEE-based Validation KLEE can be configured to exit when it encounters a specific type of error and record the test case required to reach it. Given our use case, we configure it to exit when an assertion failure is encountered. We utilise the `klee_make_symbolic()` command to mark the variables where we require non-determinism. KLEE will try different values of these variables to increase code coverage. We kill paths with unsatisfied premises by marking them with `klee_silent_exit()`. Similar to CBMC, KLEE is capable of exhaustive exploration provided that all paths are finite. Since our program has a final state, this is guaranteed.

In contrast with CBMC, KLEE utilises coverage information and, therefore, will prioritise generating test cases for paths that cover new code. This prioritisation means that KLEE has the potential to uncover paths that lead to assertion failures faster and terminate sooner. This means that the program transformation defined in Section II-B has the potential to aid the execution of the KLEE based simulation.

Listing 8. Adaptation of the x86 memory model suitable for libFuzzer, Centipede and AFL++

```
extern "C" int LLVMFuzzerTestOneInput(
  const uint8_t *data, size_t size) {
  // Additional checks to determine if
  // size is not too small
  action_list = data[MAX_ACTIONS];
  thread_list = *data + MAX_ACTIONS;
  for (int i = 0; i < MAX_ACTIONS; i++){
    int thread =
      thread_list[thread_index++] % THREAD_COUNT;
    Action action =
      action_list[action_index++] % ACTION_COUNT;
    switch (action){
      case CPU_THREAD:
        if (thread_ops[thread].empty()) continue()
    ;
        Operation op = thread_ops[thread].pop();
        if (op.type == WRITE) {
          write_to_buffer(thread, op.var, op.val);
        }
        if (op.type == READ) {
          read_buffer_or_memory(thread, op.var);
        }
        break;
      case FLUSH_BUFFER:
        if (buffer[thread].empty()) continue();
        flush_buffer(thread);
        break;
    }
  }
  check_litmus_test();
```

KLEE for x86: Recall the code snippet from Listing 3 and note the updates in Listing 7 required for KLEE. The `thread` and `action` variables are declared symbolic and chosen at each simulation step. If the premises of the semantics are not met, the simulation exits using `klee_silent_exit()`. After the end of each simulation, the conditions that check if the assertion holds are verified.

Fuzzer-based Validation Basic knowledge of the fuzzers is required to understand the appropriate compile flags and input format. Among the fuzzers that we investigate, libFuzzer and Centipede operate with in-process arrays of bytes, whereas AFL++ typically takes input as a command-line argument. As explained earlier, we opted to utilise a driver that enables AFL++ to receive inputs in the same manner as libFuzzer and Centipede. We achieved this by employing a straightforward wrapper from the Chromium project. [32]. The array of bytes is then processed to drive the model.

Fuzzers for x86: Recall the code snippet from Listing 3 and note the updates in Listing 8 required for the fuzzers. The fuzzers provide as input to the system an array called `data` and its `size`. We partition the `data` array into two arrays: one that keeps the list of actions and one that keeps the list of threads. We add an assertion to check that the number of actions and threads is sufficiently large enough for our system. We use simple if-statements to verify if the premises of an action are valid.

III. First Case Study: x86

We now present and discuss our first case-study, relating our results to research questions **RQ1** and **RQ2** identified in Section I. By implementing the x86 memory model according to the outline presented in Section II-B, and using the CBMC, KLEE, libFuzzer, Centipede and AFL++ tools described in Section II-E, we can determine their viability in uncovering weak behaviours and their effectiveness relative to a state-of-the-art simulator.

TABLE I
A Comparison of the Tools Available, Their Underlying Technique, Memory Model Implementation, Potential to Validate Disallowed Behaviours and Their Utilisation of Coverage Information

Tool	Technique	Model	Exhaustive	Coverage
RMEM [7]	enumeration	Lem	✓	✗
RMEM [7]	random	Lem	✗	✗
Naïve	fuzzing	C	✗	✗
CBMC [15]	SAT	C	✓	✗
KLEE [16]	SAT	C	✓	✗
libFuzzer [17]	fuzzing	C	✗	✓
Centipede [18]	fuzzing	C	✗	✓
AFL++ [27]	fuzzing	C	✗	✓

A. Experimental Setup

We run our experiments on an Intel Xeon CPU E5-2640 with 32GB RAM, under Ubuntu 20.04. We use RMEM version 0.1, CBMC version 5.11 and KLEE version 2.3. The fuzzers utilised are AFL++ version 4.07, the libFuzzer deployed with clang 12 and Centipede version 1.0.0. The SAT engine used by KLEE is STP version 2.3.3 and the one used by CBMC is MiniSat version 2.2.1. In pilot experiments with different solvers we observed slight differences in execution times but not significant enough to impact the comparisons between the tools. For this reason, for each tool, we use the default solver.

As KLEE has many configuration options, we asked the KLEE development team for advice on a configuration that was likely to be suitable for our setting. They advised configuring KLEE to use depth-first-search since our litmus tests have a finite search space and also advised enabling non-forking mode since the SAT expression handling does not dominate the execution. Similarly, for RMEM, the developers advised us to enable the option that enable the eager taking of transitions. RMEM needs more documentation to clarify why this option is not enabled by default and the possible cost of utilising it. For completeness, we include results with and without this option enabled.

Table I summarises the tools that we experiment with, alongside some of their characteristics. We start our experiments by exploring the strengths of RMEM which can be considered the state-of-the-art tool at the moment and implements the memory model in a custom semantics language called Lem [34]. RMEM can run in either exhaustive mode or random mode but can only prove that a behaviour is disallowed when run in exhaustive mode. Unfortunately, RMEM does not stop when it uncovers the behaviour of interest, as do all the other tools at our disposal. This means that RMEM does not report that a behaviour is allowed until it has finished exhaustive exploration, and we found that it was not trivial to temporarily modify the tool to change this.

As a sanity check, for our C models, we first utilise a naïve fuzzer, which randomly explores paths through the program's execution, similar to how the random version of RMEM does.

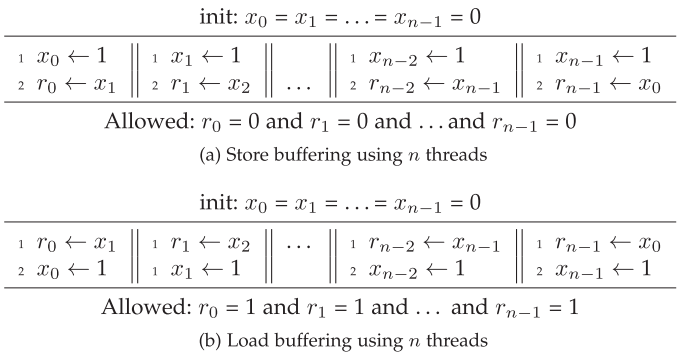

Fig. 3. Store buffering and load buffering for n threads.

Afterwards, we focus on the off-the-shelf CBMC, KLEE, libFuzzer, Centipede and AFL++ tools. All tools have the potential to uncover allowed behaviours, but only RMEM, CBMC and KLEE can prove that a behaviour is not possible (because they consider all executions). The coverage-guided fuzzers and KLEE use coverage information to guide their search.

To benchmark our approach, we use litmus tests that generalise to any number of threads. In Fig. 3, we present the generalisation of two such tests. In Fig. 3(a) we have the *allowed* store-buffering litmus test while in Fig. 3(b) we have the *disallowed* load-buffering litmus test. These tests enable us to vary the number of threads and observe how the different simulators handle them.

B. Determining the Viability of Our Approach

Given the C implementation of the x86 model, the set of program analysis tools and the litmus test, we can recall and answer our first research question:

RQ1 Can reducing the problem of memory model simulation to the analysis of a C program yield competitive performance compared with bespoke simulators?

We take each litmus test for different numbers of threads and try to determine if the reordering they describe is possible using all the tools at our disposal. We run each test/simulator combination ten times and create a box plot of the uncovered executions. We set a timeout of 2 hours per execution. Fig. 4 shows the time in seconds required to uncover the reorderings of the store buffering litmus test and Fig. 5 shows the time required to prove that the load-buffering effect is not possible. As mentioned in Table I, some tools cannot be used for exhaustive exploration and therefore Fig. 5 does not include results for these tools.

Recall from Section III-A that RMEM has many possible options that we can optionally enable during simulation. Out of all these options, the developers have advised to explore the option that makes the simulator "eagerly take transitions that do not affect observable states". This option indeed has the highest impact on performance.

For the allowed store-buffering experiments, the naïve fuzzer was only able to uncover executions for the trivial case of two threads and performed poorly in all other cases. RMEM was able to handle all the test cases we have provided and fared significantly better in the exhaustive mode but only when the option to eagerly take transitions was enabled. KLEE unfortunately timed out for larger executions, and while CBMC was able to handle all of them, it was at a significant performance overhead. libFuzzer was the fastest tool we have explored, with Centipede showing similar but slightly slower performance, and AFL++ exhibiting slower performance still, and higher runtime variance, yet still being able confirm allowed behaviours in the store buffering litmus test with 5 threads. We expect that the slower performance of Centipede and AFL is in part due to them working out-of-process, while libFuzzer is in-process (see Section II.C.3).

In contrast, the naïve fuzzer does not perform well at all, only scaling to a 2-threaded litmus test. This highlights the importance of the coverage feedback signal that the coverage-guided fuzzers exploit.

For the disallowed load-buffering case, the choice of tools is significantly lower, and only the optimised version of RMEM and CBMC were able to uncover the executions. While RMEM does scale better than CBMC in such cases, recall that it was developed over four years and already contains more than 60k lines of OCaml code, while the C implementations that we feed to the off-the-shelf simulators have less than 2k lines of C code.

Finding from RQ1. For confirming executions, off-the-shelf tools show real promise, while for showing that executions are not allowed, our results show that off-the-shelf tools are capable only for tests with small thread counts. However, this is only when specific optimisations are enabled in RMEM, and engineering those optimisations is likely a non-trivial investment.

C. Comparing the Tools

Having seen that our approach of leveraging off-the-shelf tools can be used to simulate operational memory models, we move to **RQ2**, which can be subdivided as follows:

RQ2 Of the variety of C analysis tools that we consider, which are most effective for the memory model simulation approach that we propose?
- (a) How is performance influenced by the test case size under simulation?
- (b) How is performance influenced by whether the behaviour associated with the test-case is allowed according to the memory model?

Regarding **RQ2a**, the small litmus tests that feature only two threads are solved by all techniques. However, the more heavyweight SAT-based analysis performed by CBMC puts it as a disadvantage, due to the overhead of solving a SAT query reflecting a fully-unwound program. However, we can observe that the simpler methods are not always capable of uncovering the complex executions involving more threads. The naïve fuzzer is not able to handle tests featuring more than two threads, and KLEE does not scale to the five-threaded case. In this set of experiments, the only tools that are not significantly affected by the size of the litmus test are libFuzzer and Centipede. However, we show in Section IV-D that the execution time associated with coverage-guided fuzzing does

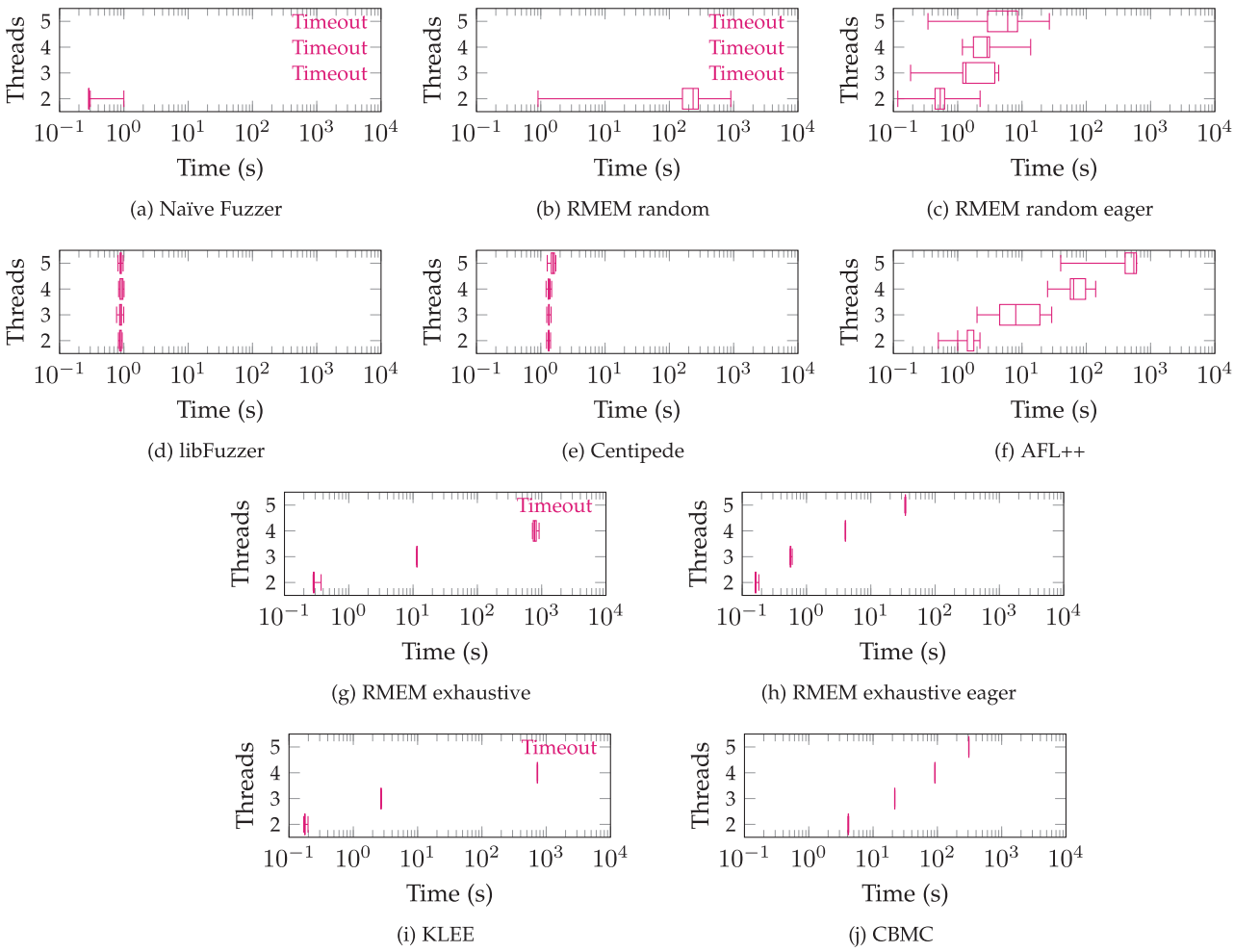

Fig. 4. Analysis times for an *allowed* litmus test (store buffering) using all tools.

increase when we apply it to a more complex memory model with larger associated litmus tests.

Finding from RQ2a. While it makes little difference which analysis tool we deploy for small litmus tests, that is no longer the case for larger ones. Out of all the off-the-shelf tools we have explored, it is advisable to use libFuzzer or CBMC. While libFuzzer has been shown to be extremely effective at discovering allowed executions, unlike CBMC it cannot show that executions are disallowed. In a context where the allowed/disallowed status of a litmus test is not known and there is no bespoke simulator, running both techniques in parallel would be advisable.

Since not all tools can prove that a behaviour is not allowed, we can only answer **RQ2b** for the ones that do. The tools have to consider all execution paths, and as a result, they require more time to do so. Having to explore all paths means that coverage information does not help. If we consider only off-the-shelf tools, CBMC is the best choice.

Finding from RQ2b. Although a heavyweight solution for simple test cases, the SAT-based approach of CBMC pays off for larger, more complex test cases when behaviours are *disallowed*.

D. Comparison With Recent SV-COMP Winners

As discussed in Section II-C, we selected CBMC, KLEE and a variety of coverage-guided fuzzing tools as they represent a range of approaches from fully symbolic (CBMC), through mixed dynamic/symbolic (KLEE), to fully dynamic (the fuzzers). However, many other analysis tools for C programs exist and could be tried. In particular, the Competition on Software Verification (SV-COMP) [35] provides a source of many analysers that target C.

To experiment with this, we selected the three winners of the "Falsification Overall" category at SV-COMP 2013 [36]. This represents tools that excel at bug-finding, and thus might be effective at confirming allowed memory model behaviours in our context. The relevant tools are Bubaak [37] (winner), PeSCo-CPA [38] (runner up) and CPAchecker [39] (third place). Bubaak is a portfolio verifier that runs several program analyses in parallel and uses runtime monitoring to coordinate them. CPAchecker is a verifier based on the notion of configurable program analysis: it features several different analysis modes (including bounded model checking and predicate analysis) implemented within a common framework. The PeSCo-CPA tool is based on CPAchecker and uses heuristics to predict which sequential combination of CPAchecker

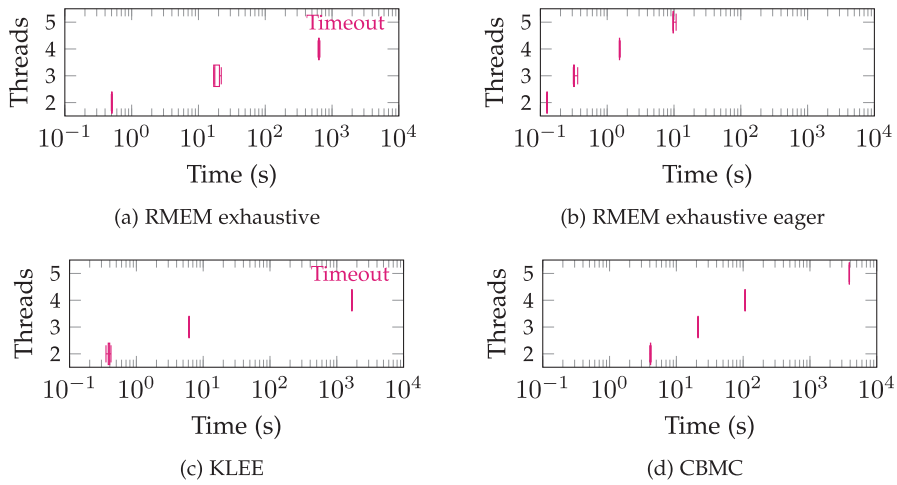

Fig. 5. Analysis times for a *disallowed* litmus test (load buffering) using those tools that are capable of exhaustive search.

configurations is likely to work best for the verification task at hand, and then attempts verification using the chosen sequence of configurations.

We used the versions of these tools provided with the artifact accompanying the SV-COMP 2023 report [40], and we liaised with the SV-COMP organisers to check that we were executing the tools in the same manner as they were executed when participating in the competition. We ran experiments using the platform described in Section III-A.

Before applying these tools, we adapted our test harness to use SV-COMP built-in functions and recommended approaches for modelling "assert" and "assume" statements, as well as non-deterministic choice. The required changes were minimal, being similar in scale to the adaptations shown in Listings 6 to 8 and for CBMC, KLEE and the coverage-guided fuzzers, respectively.

We applied the SV-COMP winners to the store-buffering litmus tests, using a timeout of 2 hours. The results for Bubaak are shown in Fig. 7: notice that it was not able to solve the 5-threaded version of the store buffering test within the timeout period. Performance for PeSCo-CPA and CPAchecker was worse still: these tools did not manage to process even the 2-threaded store buffering litmus test within 2 hours.

We also tried these SV-COMP "Falsification Overall" winners on litmus tests associated with the X+F memory model that we discuss in Section IV. We found that none of the tools—including Bubaak—were able to successfully analyse any litmus tests demonstrating allowed behaviour.

In response to our findings, the SV-COMP organisers encouraged us to submit example programs from our study for including in the SV-COMP benchmark suite, so that they can serve as challenge examples for C analysis tools. We have opened a merge request contributing a number of benchmarks, which has been approved by the SV-COMP maintainers and merged [41].

IV. Second Case Study: A CPU/FPGA Memory Model

We now describe our experience applying the approach we present in this paper to our previous work, where we have developed a more complex model for a hybrid CPU/FPGA system. Memory models for which there are no bespoke simulators, such as RMEM are the primary use-cases for which we envision our work. After introducing the system (Section IV-A) we present our experimental setup (Section IV-B), our analysis of the impact of the test harness (Section IV-C), scalability assessment (Section IV-D), and conclude by discussing the infidelities in the X+F model that we found and fixed thanks to our approach (Section IV-E).

A. The X+F Memory Model

A recent trend in heterogeneous systems is to combine a multicore CPU with a field-programmable gate array (FPGA). These hybrid CPU/FPGA systems are of particular interest because the FPGA can be customised for a specific computationally-intensive sub-task, while the overall application is coordinated by the general-purpose CPU. We focus on the Xeon+FPGA (X+F) memory model that we proposed in prior work [21], and that was originally analysed using only CBMC for a small number of memory operations.

The memory model is illustrated graphically in Fig. 6. Conceptually, the FPGA is a separate thread that runs alongside the CPU threads. In contrast to CPU threads, which issue atomic reads, writes and fences, the FPGA breaks these operations down into *requests* and *responses*. For instance, to write to memory the FPGA issues a *write request*, containing the location and value to be written. Later, if the FPGA receives a *write response* this guarantees that the write request has entered the FPGA's memory subsystem but does *not* guarantee that it has been committed to main memory. A *fence request* can be issued to indicate that writes should be propagated to main memory, and this propagation is only guaranteed to have occurred when a corresponding *fence response* is received.

The FPGA's memory subsystem is composed of *request pools*, *upstream buffers* and *downstream buffers* as shown in Fig. 6. *Requests* and *responses* have to travel through all these components when travelling between the FPGA thread and the main memory.

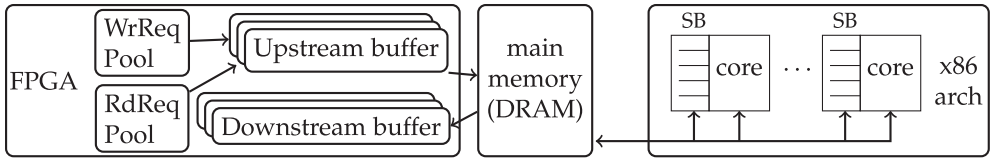

Fig. 6. A pictorial representation of the X+F memory model.

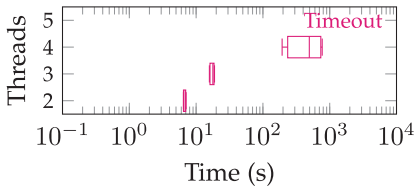

Fig. 7. Analysis times for an *allowed* litmus test (store buffering) using the Bubaak tool.

We can implement the C model of the X+F system by extending the code of the x86 system from Section II. Extending it involves adding opcodes for the FPGA thread, channel identifiers to the structure in Listing 1, and actions to the enumeration in Listing 2. The actions should correspond to the FPGA thread, *request pools*, *upstream buffers* and *downstream buffers*. Each of these elements has a separate data structure that takes into account its ordering guarantees (e.g. FIFO order for the upstream buffer).

Furthermore, in our previous work, we provided an axiomatic model for the X+F system and implemented it in Alloy. We can leverage Alloy to generate allowed and disallowed executions of different sizes that can be converted to the format our operational model can process. The size of the litmus test is described by the number of events it contains and these events represent different operations of the CPU or FPGA. Furthermore, the axiomatic model can also be configured to generate non-trivial disallowed executions where every event is 'critical' [42]. This means that removing any single event from the execution makes the execution become allowed. We can use this system to create benchmarks of different sizes and complexity for the operational simulators.

B. Experimental Setup

We reuse the same configuration described in Section III. We generate allowed executions of different sizes from the axiomatic model and run them on the operational model. Since a large number of executions can potentially be generated, we limit our exploration only to the non-trivial disallowed executions. We can then create a set of non-trivial *allowed* executions by removing fence operations from these disallowed executions.

C. Impact of Test Harness

We start by adapting the test generator to enable the alternative options described in Section II-D. These options involve encoding the litmus test for more coverage points (Listing 4) or fewer program paths (Listing 5). The alternative versions of the litmus test can be sent to the different tools to evaluate their impact on performance. We generate traces with events ranging from 6 up to 9. We recall our research question:

RQ3 How does the manner in which the memory model and litmus test are encoded as a C program impact the performance of the different tools?

Fig. 8 shows how the encoding affects the tools. The graphs show the percentage of executions uncovered in a certain amount of time. We have set a limit of 2000 seconds for each execution and run them using the different encodings and tools.

We can see that libFuzzer and Centipede, and to a lesser extent AFL++, take advantage of the coverage points provided by the first encoding of the litmus tests. Furthermore, these results corroborate the results of Section III where we have seen the poor performance of the naïve fuzzer. This result highlights the importance of coverage in fuzzing tools in uncovering the transitions that expose the behaviour needed by the litmus test.

KLEE is not significantly affected by the encoding. At the point in the program where we add branching, KLEE will have resolved all non-determinism that can affect them. These branches are resolved deterministically and do not contribute to further path explosion on a per-path basis. Furthermore, we have also empirically confirmed that other variations of this encoding that did not affect the execution time of KLEE.

CBMC similarly is not significantly impacted by the encoding utilised and the size of the formula does not significantly change.

Regardless of the choice of encoding, for KLEE and CBMC we observe a "staircase" pattern in execution time. This pattern results from the different number of events in the executions. All the tests with the same event count have a similar solving time, and each vertical jump in the staircase corresponds to moving to a new batch of tests that have more events.

Finding from RQ3. The coverage-guided fuzzers benefit from the extra coverage points added to the litmus tests. KLEE is unaffected: it does not suffer from additional path explosion because all non-determinism has already been resolved before reaching the branches. Similarly, CBMC is unaffected because the encoding does not substantially change the size and complexity of the formula that it solves.

D. Scaling up Simulation

Section IV-C shows that encoding the litmus tests for higher coverage benefits all of the coverage-guided fuzzing tools, and has no noticeable performance impact on KLEE or CBMC. We thus adopt this encoding in experiments where we explore the scalability of the tools. We recall our research question:

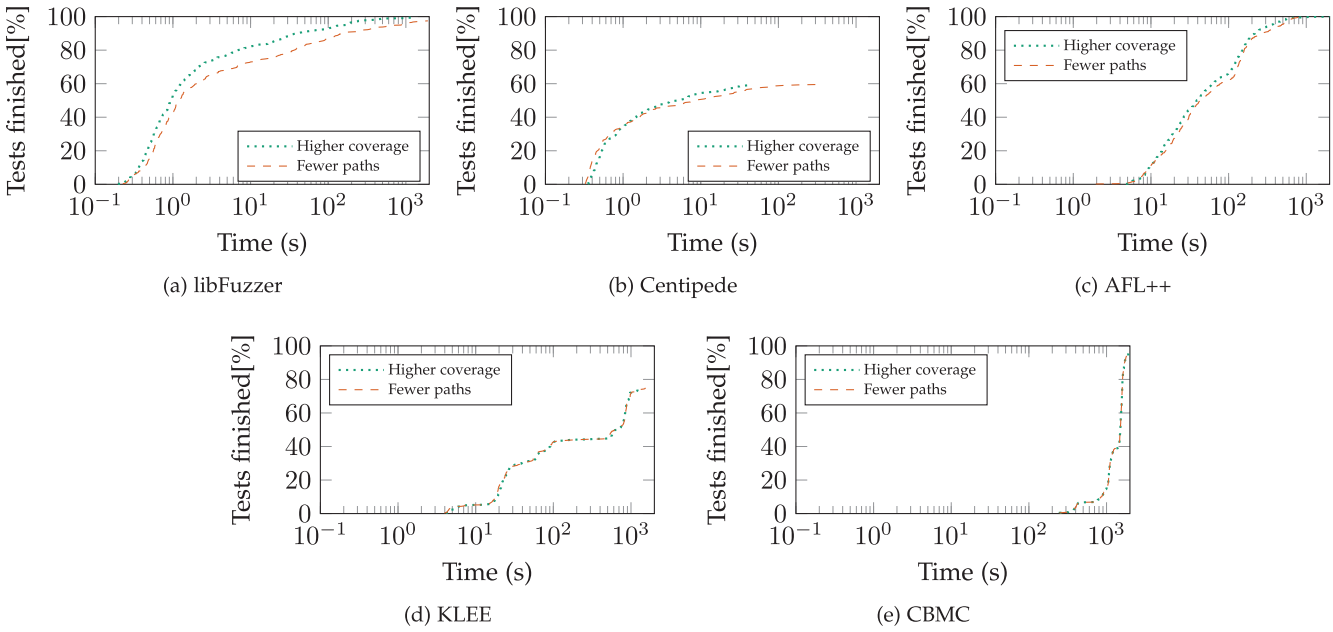

Fig. 8. The percentage of executions uncovered in a certain amount of time using the two encodings for the litmus tests.

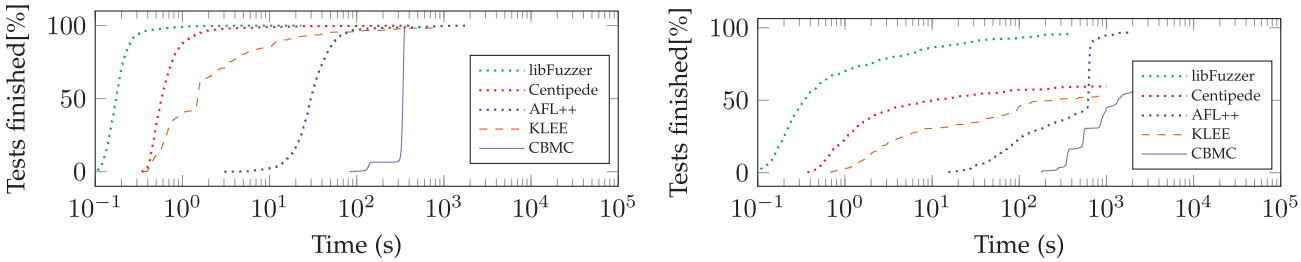

Fig. 9. The percentage of executions uncovered under a certain amount of time using different tools. All tools were able to uncover the allowed executions in (a). However, when scaling the number of events in (b) this was no longer the case: out of 350 executions, libFuzzer uncovered 335 executions, AFL++ uncovered 323, Centipede uncovered 199, CBMC uncovered 202, and KLEE uncovered only 189.

RQ4 Can our approach allow more in-depth analysis of the X+F memory model, allowing it to be better validated against its axiomatic counterpart?

Recall from Section IV-A, that Alloy allows us to generate executions of different sizes. We start with small executions, with a small number of operations per thread, that should not challenge any of the simulators. Therefore, we limit each execution to a maximum of five events, which allows us to generate a total number of 2481 allowed executions. To ensure that the tools do not hang, we limit the total execution time to 500 seconds. After verifying that the performance of the tools translates to this model, we generate executions with more operations to evaluate scalability. We generate 50 executions for each thread size from 6 up to 16 operations, summing up to 350 executions. We also increase the timeout from 500 seconds to 2000 seconds.

Fig. 9 shows the percentage of executions that could be shown to be allowed after a certain amount of time. Fig. 9(a) shows the results for a small number of events (maximum 5 events). In this case, all tools were able to find the path through the program that led to the reordering required by all 2481 executions. However, this is no longer the case when scaling the number of events: Fig. 9(b) shows results for executions containing up to 16 events. Out of 350 executions, libFuzzer uncovered 335 executions, AFL++ uncovered 323 executions, Centipede uncovered 199, CBMC uncovered 202, and KLEE uncovered only 189. Recall that these executions were generated from the X+F axiomatic memory model, and thus they are all expected to be valid executions. Thus the number of executions that each tool was *unable* to verify represents a likely limitation of the tool.

The libFuzzer and Centipede tools significantly outperform CBMC and KLEE, managing to uncover the path that leads to the reorderings faster than the other tools. The performance of AFL++ lies between that of the other two fuzzers and KLEE. The time required by CBMC depends on the test size and explains the staircase pattern that we see in both figures. However, despite its slow execution, CBMC did not get stuck like the fuzzers and KLEE sometimes did, and was the only tool that could uncover some of the more challenging executions. It can thus be considered the slowest yet most reliable tool.

Our strategy for fast model evaluation. We can assume that every given set of test-cases will be composed of challenging and easy executions. Our experiments show that coverage-guided fuzzers can quickly confirm that simple executions are allowed but fails with some of the more challenging ones. On the other hand, CBMC is able to confirm that all the executions we considered in our experiments were allowed, if the increased simulation time is accepted, but requires a significant amount of time to do so. We can therefore come up with a strategy where for any given set of executions, we first run e.g. libFuzzer to uncover the easy ones and only utilise CBMC for the more challenging ones. We discuss in Section IV-E how this technique has allowed us to fix some infidelities in the model.

Finding from RQ4. Deploying both coverage-guided fuzzing and CBMC enables feasible exploration of the model for a considerably higher number of events.

Tool limitation. For some executions, KLEE is very fast, while for others, it does not terminate within 24 hours. We sent sample test cases to the developers, and they informed us that these cases revealed a limitation of the tool. KLEE compares each new query with previous solved ones to reduce execution time. However, some executions create large expressions that KLEE is unable to evaluate, so it gets stuck. The developers are currently working on a fix.

E. Fixing the CPU/FPGA Model

In our prior work, we only used CBMC and were limited by how many litmus tests we could run. By adding coverage-guided fuzzers to our arsenal of tools, we can simulate with much higher throughput by only using CBMC as a fallback or as a sanity check for the rare cases where we find something unexpected. We do not strictly need to use CBMC, but it serves as means of cross-checking the results of the fuzzers.

In our experiments, almost all the allowed executions were proven so by the fuzzer. We switched to CBMC for a second option for the ambiguous tests. When not even CBMC could find a path through the program that leads to the required reordering, we manually inspected the executions. We realised that these reorderings were impossible and that the axiomatic and operational models do not perfectly match. This mismatch led us to some infidelities in the original model that we were able to fix.

As a result of this work, four axioms in our original work have been modified. We list the complete set of corrected axioms in Fig. 10. All of the relations mentioned in the axioms have the same definitions as in our prior work [21], with the additional definition of $\texttt{fencepair}$ as $\texttt{fenceonepair} \cup \texttt{fenceallpair}$. It is beyond the scope of this article to properly explain all the definitions used in the axioms, so we refer the interested reader to our prior work [21]. Below, we outline the effect of our changes.

The OBSERVE-SAME-CHANNEL axiom was originally an irreflexivity axiom that prevented writes from different threads

$$\textbf{acyclic}((\texttt{poloc} \cup \texttt{rf} \cup \texttt{fr} \cup \texttt{co}) \cap \texttt{CPU}^2) \quad \text{SC-PER-LOC}$$

$$\textbf{acyclic}(\texttt{ppo} \cup \texttt{fence} \cup \texttt{rfe} \cup \texttt{fre} \cup \texttt{co}) \quad \text{PROPAGATION}$$

$$\textbf{irreflexive}(\texttt{fr} ; \texttt{poch} ; \texttt{readpair}) \quad \text{READ-AFTER-WRITE}$$

$$\textbf{irreflexive}(\texttt{fr} ; \texttt{poFnRsp} ; \texttt{po} ; \texttt{readpair}) \quad \text{READ-AFTER-FENCE}$$

$$\textbf{irreflexive}(\texttt{rf} ; \texttt{po}) \quad \text{NO-READ-FROM-FUTURE}$$

$$\textbf{acyclic}(\texttt{fre} \cup \texttt{rfe} \cup (\texttt{rf} \setminus \texttt{sch}) \cup \texttt{poch} \cup \texttt{ppoCPU}) \quad \text{OBSERVE-SAME-CHANNEL}$$

$$\textbf{irreflexive}(\texttt{po} ; \texttt{fencepair} ; \texttt{po} ; \texttt{writepair}^{-1}) \quad \text{FENCE-RESPONSE}$$

$$\textbf{irreflexive}(\texttt{po} ; (\texttt{fencepair} \cup \texttt{writepair}) ; \texttt{po} ; \texttt{fenceallpair}^{-1}) \quad \text{FENCE-BLOCK}$$

$$\textbf{irreflexive}(\texttt{rf} ; \texttt{poloc} ; \texttt{co}) \quad \text{WRITE-ORDER}$$

Fig. 10. The revised axioms of the X+F memory model, with modifications highlighted.

from being observed out-of-order on the same channel. It has now been extended to disallow this behaviour for reads and fences, not merely for writes. The FENCE-RESPONSE and FENCE-BLOCK axioms were present in our original article; they describe the orderings that fences enforce on the executions. These axioms have been extended to take into account how fences can block additional writes from propagating. The WRITE-ORDER axiom is a new axiom, added to ensure that multiple writes to the same location on the FPGA are propagated in order.

V. Related Work

Operational memory models have been proposed for widely used memory models including x86 [12], [43], POWER [9], ARM [3], [4] and RISC-V [5], and **model checkers** have been widely used to simulate them. For instance, Alglave et al. [44] showed how to transform the problem of verification under a weak memory model into the problem of verification under SC, by transforming each memory access in the program so that it explicitly manipulates a buffer rather than main memory directly. The verification-under-SC problem can then be handled by an off-the-shelf model checker; Alglave et al. [44] use CBMC, while Still and Barnat [45] use DIVINE. In both cases, the approach differs from ours because it relies on the design of program transformations that correspond to specific memory models (x86, PSO, RMO, and POWER), while we show how to encode the transition system of an arbitrary memory model directly in C. This makes our approach more suitable for a memory model that is still in active development, whose transition system may still be in flux, and which is not yet sufficiently well understood to be confidently translated into equivalent program transformations. Besides, it is not clear how some of the more complicated memory models, like the X+F model we considered, can be recast as program transformations; indeed, Lahav and Vafeiadis [46] have shown that some memory models, notably ARM's, *cannot* be explained solely using program transformations.

Program transformations are popular to help fuzzing tools find more bugs. For instance, previous work [47] has proposed

semantics-changing transformations that remove checks that fuzzing is struggling to get past. Armstrong et al. [33] proposed LLVM passes that increase code coverage. While these techniques can be applied to general programs, in our case, program transformations only help when applied to the encoding of the litmus test.

RMEM [7] is a **bespoke simulator** for a variety of operational memory models [3], [4], [48]. It is widely used, but involves a substantial amount of engineering effort to implement the desired algorithms for exploring the models (e.g. randomly, exhaustively, or in a user-guided fashion). In our approach, we outsource the exploration task to a range of off-the-shelf tools instead. *Murphi* is another simulator for operational semantics that has been used to explore GPU memory models [49] and also to verify heterogeneous cache coherency protocols [50], but again, it lacks the generality of our approach because it only supports the exploration algorithms that are hardcoded into it.

Axiomatic memory models are an alternative way of representing memory models, with numerous memory models implemented in the *CAT* language for the *herd* simulator [2]. *Alloy* is an open-source language and analyser for software modelling that has been used to simulate memory models [51]. *CDSChecker* [52] is a model-checker for C++ programs that uses several techniques to minimise the number of behaviours that need exploring. *GenMC* [53] is a stateless model checker that can efficiently verify C++ programs. *Dartagnan* [54] is another model checker that can accept any axiomatic memory model that is specified in the CAT language. Recent work by Ponce de León et al. [55] and by Fan et al. [56] has cast memory-model consistency axioms as a theory that can be deployed by SMT solvers. While axiomatic memory model simulators, such as all of those mentioned above, are generally faster than those for operational models, they lack some intuitive features such as providing execution paths that the user can step through in order to witness the reordering. It is possible in principle to convert operational models into equivalent axiomatic models, and thereby obtain fast simulation for operational models, but the process is difficult. For instance, Pulte et al. [3] developed operational and axiomatic versions of the Arm memory model, and proved them equivalent, but this was a substantial manual effort and not necessarily appropriate for a prototype model such as X+F. (On the other hand, converting axiomatic models into equivalent operational models is more straightforward, and progress has recently being made towards performing that conversion automatically by Godbole et al. [57].)

The idea of **problem reduction to a C program** has been explored in other domains; for instance, Verilator [58] is a popular open-source Verilog simulator that works by translating a Verilog design into a C program that can then be executed or otherwise analysed [59]. As discussed in Section I, coverage-guided fuzzing has been used as a model-exploration technique in domains where formal verification or symbolic reasoning techniques do not scale well, such as demonstrating the satisfiability of SMT formulas for floating-point arithmetic [20].

The trinity of CBMC, KLEE and libFuzzer have been previously shown by Priya et al. [60] to complement each other and uncover bugs in different applications.

VI. Conclusion

We have investigated how operational memory models can be simulated by reducing the decision problem of "whether a given operational model allows a given program behaviour" to the decision problem of "whether a given C program is safe", which can be handled by a variety of off-the-shelf tools. This has allowed us to evaluate five such tools: a model checker (CBMC), a symbolic analysis tool (KLEE) and three coverage-guided fuzzers (libFuzzer, Centipede and AFL++), comparing them to a bespoke simulator (RMEM).

The key take-away from our experience is that we highly recommend that researchers and engineers interested in operational memory model simulation consider our "reduction to C" approach, because it lifts the burden of having to implement a variety of analysis algorithms in a bespoke tool. Our experience is that coverage-guided fuzzing, and the libFuzzer tool in particular, shine when it comes to fast analysis of allowed behaviours. By experimenting with three different coverage-guided fuzzers—libFuzzer, Centipede and AFL++, all of which yield good results, we provide evidence that coverage-guided fuzzing in general (rather than one tool in particular) may be a good fit for this problem domain.

The symbolic CBMC tool is effective at exhaustive exploration of reasonably large litmus tests. Our second case study, on a CPU/FPGA memory model, showcases the value of our approach by enabling several discrepancies in an axiomatic description of this memory model to be found and fixed.

A potential concern regarding our findings about fuzzing is whether this finding is future-proof: as fuzzer developers, who are primarily focused on security issues, make changes to their tools, might these changes have a *negative* effect on tool effectiveness in our domain? We believe this is unlikely, since to detect vulnerabilities a tool must excel at driving execution towards all parts of the program under test (since it is not known, a priori, where vulnerabilities may lie). By encoding memory model analysis as a reachability problem, our approach should benefit from advances in fuzzing technology.

Our experiments with various verification tools that performed well in the "Falsification Overall" category of the SV-COMP 2023 competition showed that these tools do not scale to our benchmarks. In response to this we have submitted example benchmark programs to the SV-COMP benchmark suite, so that they can be used as challenge examples for verification tools in the future.

The SV-COMP verification tools are attractive for our purpose as they can be directly applied to C programs. It could also be interesting to look at applying other model checking tools, such as the SPIN model checker [61]. However, SPIN has its own input language, Promela, so leveraging SPIN would require encoding our memory models in this specialised language, which would be contrary to our aim of leveraging off-the-shelf tools for C. While there has been work on extracting Promela

models from C programs [62] or translating C programs into Promela [63], these approaches are not currently maintained, and are not "push button": they require customisation of the extracted or generated code due to the large semantic gap between C and Promela. Many other model checking tools that have their own custom input language would be hard to apply in our domain for the same reason.

Acknowledgment

The authors thank Frank Busse, Christopher Pulte, and Ben Simner for their help with RMEM and KLEE. The authors are grateful to Dirk Beyer and Philipp Wendler for advice related to SV-COMP and for reviewing our benchmark submissions. The authors thank the anonymous reviewers for their useful feedback which has helped to improve the paper.

Data Availability and Open Sourcing

An artifact containing the source code, scripts and data associated with this paper is openly available.[1]

References

[1] K. Rupp, "40 years of microprocessor trend data," 2015. Accessed: Nov. 2, 2023. [Online]. Available: https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data

[2] J. Alglave, L. Maranget, and M. Tautschnig, "Herding cats: Modelling, simulation, testing, and data mining for weak memory," *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 2, pp. 1–74, Jul. 2014. [Online]. Available: https://doi.org/10.1145/2627752

[3] C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, and P. Sewell, "Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 1–29, Dec. 2017. [Online]. Available: https://doi.org/10.1145/3158107

[4] B. Simner et al., "ARMv8-A system semantics: Instruction fetch in relaxed architectures," in *Proc. 29th Eur. Symp. Program. (ESOP)*, Dublin, Ireland: ACM, Mar. 2020, pp. 1–30. Accessed: Nov. 2, 2023. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-44914-8_23

[5] C. Pulte, J. Pichon-Pharabod, J. Kang, S.-H. Lee, and C.-K. Hur, "Promising-ARM/RISC-V: A simpler and faster operational concurrency model," in *Proc. 40th ACM SIGPLAN Conf. Program. Lang. Des. Implementation (PLDI)*. New York, NY, USA: ACM, 2019, pp. 1–15. [Online]. Available: https://doi.org/10.1145/3314221.3314624

[6] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "Litmus: Running tests against hardware," in *Tools and Algorithms for the Construction and Analysis of Systems*, P. A. Abdulla and K. R. M. Leino, Eds., Berlin, Heidelberg: Springer, 2011, pp. 41–44.

[7] A. Armstrong et al., "RMEM: A tool for exploring relaxed-memory concurrency," Cambridge, U.K., 2022. Accessed: Nov. 2, 2023. [Online]. Available: https://github.com/rems-project/rmem

[8] S. Flur et al., "Modelling the ARMv8 architecture, operationally: Concurrency and ISA," *SIGPLAN Not.*, vol. 51, no. 1, pp. 608–621, Jan. 2016. [Online]. Available: https://doi.org/10.1145/2914770.2837615

[9] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams, "Understanding power multiprocessors," in *Proc. 32nd ACM SIGPLAN Conf. Program. Lang. Des. Implementation (PLDI)*. New York, NY, USA: ACM, 2011, pp. 175–186. [Online]. Available: https://doi.org/10.1145/1993498.1993520

[10] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell, "Clarifying and compiling C/C++ concurrency: From C++11 to POWER," in *Proc. 39th Annu. ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*. New York, NY, USA: ACM, 2012, pp. 509–520. [Online]. Available: https://doi.org/10.1145/2103656.2103717

[11] S. Sarkar et al., "Synchronising C/C++ and power," in *Proc. 33rd ACM SIGPLAN Conf. Program. Lang. Des. Implementation (PLDI)*. New York, NY, USA: ACM, 2012, pp. 311–322. [Online]. Available: https://doi.org/10.1145/2254064.2254102

[12] S. Owens, S. Sarkar, and P. Sewell, "A better x86 memory model: x86-TSO," in *Theorem Proving in Higher Order Logics*, S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds., Berlin, Heidelberg: Springer, 2009, pp. 391–407.

[13] A. De, A. Roychoudhury, and D. D'Souza, "Java memory model aware software validation," in *Proc. 8th ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools Eng. (PASTE)*. New York, NY, USA: ACM, 2008, pp. 8–14. [Online]. Available: https://doi.org/10.1145/1512475.1512478

[14] A. Roychoudhury and T. Mitra, "Specifying multithreaded Java semantics for program verification," in *Proc. 24th Int. Conf. Softw. Eng. (ICSE)*. New York, NY, USA: ACM, 2002, pp. 489–499. [Online]. Available: https://doi.org/10.1145/581339.581399

[15] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (Lecture Notes in Computer Science), K. Jensen and A. Podelski, Eds., vol. 2988, Germany: Springer, 2004, pp. 168–176.

[16] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Conf. Operat. Syst. Des. Implementation (OSDI)*. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224.

[17] K. Serebryany, "libFuzzer: A library for coverage-guided fuzz testing," Google, 2022. Accessed: Nov. 2, 2023. [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[18] Google, *Centipede*, 2023. Accessed: Nov. 2, 2023. [Online]. Available: https://github.com/google/fuzztest/tree/main/centipede

[19] Google, *American Fuzzy Lop*, 2022. Accessed: Nov. 2, 2023. [Online]. Available: https://github.com/google/AFL

[20] D. Liew, C. Cadar, A. F. Donaldson, and J. R. Stinnett, "Just fuzz it: Solving floating-point constraints using coverage-guided fuzzing," in *Proc. 27th ACM Joint Meet. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*. New York, NY, USA: ACM, 2019, pp. 521–532. [Online]. Available: https://doi.org/10.1145/3338906.3338921

[21] D. Iorga, A. F. Donaldson, T. Sorensen, and J. Wickerson, "The semantics of shared memory in Intel CPU/FPGA systems," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1–28, Oct. 2021. [Online]. Available: https://doi.org/10.1145/3485497

[22] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. 28, no. 9, pp. 690–691, Sep. 1979. [Online]. Available: https://doi.org/10.1109/TC.1979.1675439

[23] T. Sorensen and A. F. Donaldson, "Exposing errors related to weak memory in GPU applications," in *Proc. 37th ACM SIGPLAN Conf. Program. Lang. Des. Implementation (PLDI)*. New York, NY, USA: ACM, 2016, pp. 100–113. [Online]. Available: https://doi.org/10.1145/2908080.2908114

[24] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Proc. 6th Int. Conf. Theory Appl. Satisfiability Test., SAT, Sel. Rev. Papers*, in Lecture Notes in Computer Science, E. Giunchiglia and A. Tacchella, Eds., Santa Margherita Ligure, Italy, vol. 2919, Springer, May 5–8, 2003, pp. 502–518. [Online]. Available: https://doi.org/10.1007/978-3-540-24605-3_37

[25] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2023. Accessed: Nov. 2, 2023. [Online]. Available: https://www.fuzzingbook.org/

[26] P. A. Vikhar, "Evolutionary algorithms: A critical review and its future prospects," in *Proc. Int. Conf. Global Trends Signal Process., Inf. Comput. Commun. (ICGTSPICC)*, 2016, pp. 261–265.

[27] A. Fioraldi, D. C. Maier, H. EiSSfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *Proc. 14th USENIX Workshop Offensive Technol. (WOOT)*, Y. Yarom and S. Zennou, Eds., USENIX Association, Aug. 11, 2020. Accessed: Nov. 2, 2023. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi

[28] AFL++ Project, *AFL++ Overview*, 2023. Accessed: Nov. 2, 2023. [Online]. Available: https://aflplus.plus/

[29] Advanced Fuzzing League ++, *American Fuzzy Lop plus plus (AFL++)*, 2023. Accessed: Nov. 2, 2023. [Online]. Available: https://github.com/AFLplusplus/AFLplusplus

[30] A. Arya, O. Chang, M. Moroz, M. Barbella, and J. Metzman, "Open sourcing clusterfuzz," 2019. Accessed: Nov. 2, 2023.

[1] https://zenodo.org/records/10067778

[Online]. Available: https://opensource.googleblog.com/2019/02/open-sourcing-clusterfuzz.html

[31] Google, "OSS-Fuzz," 2023. Accessed: Nov. 2, 2023. [Online]. Available: https://github.com/google/oss-fuzz

[32] LLVM Project, "afl_driver.cpp—A glue between AFL and libFuzzer," 2023. Accessed: Nov. 2, 2023. [Online]. Available: https://source.chromium.org/chromiumos/chromiumos/codesearch/+/main:src/third_party/llvm-project/compiler-rt/lib/fuzzer/afl/afl_driver.cpp

[33] lafintel, *Circumventing Fuzzing Roadblocks With Compiler Transformations*, Aug. 2016. Accessed: Nov. 2, 2023. [Online]. Available: https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/

[34] D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell, "Lem: Reusable engineering of real-world semantics," in *Proc. 19th ACM SIGPLAN Int. Conf. Functional Program. (ICFP)*. New York, NY, USA: ACM, 2014, pp. 175–188. [Online]. Available: https://doi.org/10.1145/2628136.2628143

[35] D. Beyer, "Competition on software verification (SV-COMP)," 2023. Accessed: Nov. 2, 2023. [Online]. Available: https://sv-comp.sosy-lab.org

[36] D. Beyer, "Competition on software verification and witness validation: SV-COMP 2023," in *Proc. 29th Int. Conf. Tools Algorithms Constr. Anal. Syst. (TACAS), Held as Part of the Eur. Joint Conf. Theory Pract. Softw. (ETAPS)*, Paris, France, Apr. 22–27, 2023, Part II, Lecture Notes in Computer Science, S. Sankaranarayanan and N. Sharygina, Eds., vol. 13994. Cham, Switzerland: Springer, 2023, pp. 495–522. [Online]. Available: https://doi.org/10.1007/978-3-031-30820-8_29

[37] M. Chalupa and T. A. Henzinger, "Bubaak: Runtime monitoring of program verifiers (competition contribution)," in *Proc. 29th Int. Conf. Tools Algorithms Constr. Anal. Syst. (TACAS), Held as Part of the Eur. Joint Conf. Theory Pract. Softw. (ETAPS)*, Paris, France, Apr. 22–27, 2023, Part II, in Lecture Notes in Computer Science, S. Sankaranarayanan and N. Sharygina, Eds., vol. 13994. Cham, Switzerland: Springer, 2023, pp. 535–540. [Online]. Available: https://doi.org/10.1007/978-3-031-30820-8_32

[38] C. Richter and H. Wehrheim, "PeSCo: Predicting sequential combinations of verifiers (competition contribution)," in *Proc. 25th Int. Conf. Tools Algorithms Constr. Anal. Syst. (TACAS: TOOLympics), Held as Part of ETAPS 2019*, Prague, Czech Republic, Apr. 6–11, 2019, Part III, in Lecture Notes in Computer Science, D. Beyer, M. Huisman, F. Kordon, and B. Steffen, Eds., vol. 11429. Cham, Switzerland: Springer, 2019, pp. 229–233. [Online]. Available: https://doi.org/10.1007/978-3-030-17502-3_19

[39] D. Beyer and M. E. Keremoglu, "CPAchecker: A tool for configurable software verification," in *Proc. 23th Int. Conf. Comput. Aided Verification (CAV)*, in Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds., Snowbird, UT, USA, vol. 6806. Springer, Jul. 14–20, 2011, pp. 184–190. [Online]. Available: https://doi.org/10.1007/978-3-642-22110-1_16

[40] D. Beyer, "Verifiers and Validators of the 12th Intl. Competition on Software Verification (SV-COMP 2023)," Feb. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7627829

[41] D. Iorga, "Merge request to add memory model tests to the SV-COMP benchmark set," 2023. Accessed: Nov. 2, 2023. [Online]. Available: https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/merge_requests/1444

[42] D. Lustig, A. Wright, A. Papakonstantinou, and O. Giroux, "Automated synthesis of comprehensive memory model litmus test suites," in *Proc. 22nd Int. Conf. Archit. Support Program. Lang. Operat. Syst. (ASPLOS)*. New York, NY, USA: ACM, 2017, pp. 661–675. [Online]. Available: https://doi.org/10.1145/3037697.3037723

[43] S. Sarkar et al., "The semantics of X86-CC multiprocessor machine code," in *Proc. 36th ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, Z. Shao and B. C. Pierce, Eds., Savannah, GA, USA, Jan. 21–23, 2009, New York, NY, USA: ACM, 2009, pp. 379–391. [Online]. Available: https://doi.org/10.1145/1480881.1480929

[44] J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig, "Software verification for weak memory via program transformation," 2012, *arXiv:*abs/1207.7264.

[45] V. Still and J. Barnat, "Model checking of C++ programs under the x86-tso memory model," in *Proc. Formal Methods Softw. Eng. - 20th Int. Conf. Formal Eng. Methods (ICFEM)*, in Lecture Notes in Computer Science, J. Sun and M. Sun, Eds., Gold Coast, QLD, Australia, vol. 11232, Cham, Switzerland: Springer, Nov. 12–16, 2018, pp. 124–140. [Online]. Available: https://doi.org/10.1007/978-3-030-02450-5_8

[46] O. Lahav and V. Vafeiadis, "Explaining relaxed memory models with program transformations," in *Proc. 21st Int. Symp. FM 2016: Formal Methods,* in Lecture Notes in Computer Science, J. S. Fitzgerald, C. L. Heitmeyer, S. Gnesi, and A. Philippou, Eds., Limassol, Cyprus, vol. 9995, Nov. 9–11, 2016, pp. 479–495. [Online]. Available: https://doi.org/10.1007/978-3-319-48989-6_29

[47] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: Fuzzing by program transformation," in *Proc. IEEE Symp. Secur. Privacy (SP)*, 2018, pp. 697–710.

[48] A. Armstrong et al., "ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 1–31, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3290384

[49] T. Sorensen, G. Gopalakrishnan, and V. Grover, "Towards shared memory consistency models for GPUs," in *Proc. 27th Int. ACM Conf. Int. Conf. Supercomput. (ICS)*, New York, NY, USA: ACM, 2013, pp. 489–490. [Online]. Available: https://doi.org/10.1145/2464996.2467280

[50] N. Oswald, V. Gavrielatos, V. Nagarajan, T. Olausson, D. J. Sorin, and R. Carr, "Heterogen: Automatic synthesis of heterogeneous cache coherence protocols," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, Piscataway, NJ, USA: IEEE, 2022, pp. 62–70.

[51] J. Wickerson, M. Batty, T. Sorensen, and G. A. Constantinides, "Automatically comparing memory consistency models," in *Proc. 44th ACM SIGPLAN Symp. Princ. Program. Lang. (POPL)*, New York, NY, USA: ACM, 2017, pp. 190–204. [Online]. Available: https://doi.org/10.1145/3009837.3009838

[52] B. Norris and B. Demsky, "CDSChecker: Checking concurrent data structures written with C/C++ atomics," *SIGPLAN Not.*, vol. 48, no. 10, pp. 131–150, Oct. 2013. [Online]. Available: https://doi.org/10.1145/2544173.2509514

[53] M. Kokologiannakis and V. Vafeiadis, "GenMC: A model checker for weak memory models," in *Computer Aided Verification*, A. Silva and K. R. M. Leino, Eds., Cham: Springer International Publishing, 2021, pp. 427–440.

[54] H. P. de León, F. Furbach, K. Heljanko, and R. Meyer, "Dartagnan: Bounded model checking for weak memory models (competition contribution)," in *Proc. 26th Int. Conf. Tools Algorithms Constr. Anal. Syst. (TACAS), Held as Part of the Eur. Joint Conf. Theory Pract. Softw. (ETAPS)*, Dublin, Ireland, Apr. 25–30, 2020, Part II, in Lecture Notes in Computer Science, A. Biere and D. Parker, Eds., vol. 12079, Springer, 2020, pp. 378–382. [Online]. Available: https://doi.org/10.1007/978-3-030-45237-7_24

[55] T. Haas, R. Meyer, and H. P. de León, "CAAT: Consistency as a theory," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, pp. 114–144, 2022. [Online]. Available: https://doi.org/10.1145/3563292

[56] H. Fan, Z. Sun, and F. He, "Satisfiability modulo ordering consistency theory for SC, TSO, and PSO memory models," *ACM Trans. Program. Lang. Syst.*, vol. 45, no. 1, pp. 6:1–6:37, 2023. [Online]. Available: https://doi.org/10.1145/3579835

[57] A. Godbole, Y. A. Manerkar, and S. A. Seshia, "Automated conversion of axiomatic to operational models: Theory and practice," in *Proc. 22nd Formal Methods Comput.-Aided Des. (FMCAD)*, A. Griggio and N. Rungta, Eds., Trento, Italy, Piscataway, NJ, USA: IEEE, Oct. 17–21, 2022, pp. 331–342. [Online]. Available: https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_40

[58] W. Snyder, "Verilator open-source SystemVerilog simulator and lint system," Veripool, 2022. Accessed: Nov. 2, 2023. [Online]. Available: https://www.veripool.org/verilator/

[59] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," 2021, *arXiv:abs/2102.02308*.

[60] S. Priya, X. Zhou, Y. Su, Y. Vizel, Y. Bao, and A. Gurfinkel, "Verifying verified code," in *Automated Technology for Verification and Analysis*, Z. Hou and V. Ganesh, Eds., Cham: Springer International Publishing, 2021, pp. 187–202.

[61] G. J. Holzmann, *The SPIN Model Checker—Primer and Reference Manual*. Reading, MA, USA: Addison-Wesley, 2004.

[62] G. J. Holzmann and M. H. Smith, "Software model checking: Extracting verification models from source code," *Softw. Test. Verification Reliab.*, vol. 11, no. 2, pp. 65–79, 2001.

[63] K. Jiang, "Model checking C programs by translating C to Promela," Master's thesis, UPPSALA Univ., 2009.

Dan Iorga received the Ph.D. degree in computer science from Imperial College London, in 2022. He is currently a Staff Engineer with Qualcomm, working on verifying ARM cores. His research interests are in testing, verification, and specification design of shared-memory multicore systems, especially heterogeneous systems.

John Wickerson (Senior Member, IEEE) received the Ph.D. degree in computer science from the University of Cambridge, in 2013. He is a Senior Lecturer with the Department of Electrical and Electronic Engineering at Imperial College London. His research interests include high-level synthesis, the design and implementation of programming languages, and software verification. He is a member of the ACM.

Alastair F. Donaldson (Member, IEEE) received the Ph.D. degree from the University of Glasgow. He is a Professor with Imperial College London, where he leads the Multicore Programming Group and serves as the Director of Research for the Department of Computing. His main research interests lie in the intersection of software testing, formal verification, and parallel computing. He was Founder and Director of GraphicsFuzz, a startup company based on his group's research on GPU compiler testing, which was acquired by Google in 2018. After the acquisition, he spent some time working as a Software Engineer with Google before returning full time to Imperial in 2021. He received the 2017 Roger Needham Award in recognition for his research achievements. He is an ACM Senior Member and a fellow of the British Computer Society.