Effective Processor Verification with Logic Fuzzer Enhanced Co-simulation

Nursultan Kabylkas
nkabylka@ucsc.edu
UC Santa Cruz
Santa Cruz, CA, USA

Tommy Thorn
tommy.thorn@esperantotech.com
Esperanto Technologies
Mountain View, CA, USA

Shreesha Srinath[*]
shreesha.srinath@intel.com
Intel
Portland, OR, USA

Polychronis Xekalakis[*]
pxekalakis@nvidia.com
Nvidia
Portland, OR, USA

Jose Renau
renau@ucsc.edu
UC Santa Cruz
Santa Cruz, CA, USA

ABSTRACT

The study on verification trends in the semiconductor industry shows that the design complexity is increasing, fewer companies achieve first silicon success and need more spins before production, companies hire more verification engineers, and 53% of the whole hardware-design-cycle is spent on the design verification [18]. The cost of the respin is high, and more than 40% of the cases that contribute to it are post-fabrication functional bug exposures [16]. The study also shows that 65% of verification engineers' time is spent on debug, test creation, and simulation [17]. This paper presents a set of tools for RISC-V processor verification engineers that help to expose more bugs before production and increase the productivity of time spent on debugging, test creation and simulation. We present Logic Fuzzer (LF), a novel tool that expands the verification space exploration without the creation of additional verification tests. The LF randomizes the states or control signals of the design-under-test at the places that do not affect functionality. It brings the processor execution outside its normal flow to increase the number of microarchitectural states exercised by the tests. We also present Dromajo, the state of the art processor verification framework for RISC-V cores. Dromajo is an RV64GC emulator that was designed specifically for co-simulation purposes. It can boot Linux, handle external stimuli, such as interrupts and debug requests on the fly, and can be integrated into existing testbench infrastructure with minimal effort. We evaluate the effectiveness of the tools on three RISC-V cores: CVA6, BlackParrot, and BOOM. Dromajo by itself found a total of nine bugs. The enhancement of Dromajo with the Logic Fuzzer increases the exposed bug count to thirteen *without* creating additional verification tests.

[*]The contribution was made while at Esperanto Technologies

MICRO-54, October 18–22, 2021, Athens, Greece
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

CCS CONCEPTS

• **Computing methodologies → Model verification and validation**; **Simulation evaluation**; **Simulation tools**; • **Computer systems organization → Architectures**.

KEYWORDS

microprocessor verification, co-simulation, enhanced simulation, RISC-V

ACM Reference Format:
Nursultan Kabylkas, Tommy Thorn, Shreesha Srinath, Polychronis Xekalakis, and Jose Renau. 2021. Effective Processor Verification with Logic Fuzzer Enhanced Co-simulation. In *The 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-54), October 18–22, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Modern microprocessors are complex systems. The study on verification trends in the semiconductor industry shows that the design complexity is increasing, fewer companies achieve first silicon success, companies hire more verification engineers, and 53% of the whole hardware-design-cycle is spent on the design verification [18].

When verifying microprocessors, the common practice is to build a *co-simulation* infrastructure [27, 28]. The co-simulation compares the design-under-test (DUT) execution against the high-level software model of the design, also known as the "golden" model. The underlying idea is simple: when we run the code on the DUT and the model, the architectural state of both must be the same at any given moment. In the case of a mismatch, reasons are investigated, and bugs are uncovered.

To get confidence about the DUT correctness, engineers use various proxy metrics, such as automatic code-based and circuit-structure-based coverage, as well as manual implementation specific functional coverage [38]. Another typical metric is to track the bugs found per week. The verification team thoroughly studies the architecture specification and carefully designs the functional coverage models. The plan must cover ample verification space and, most importantly, consider the corner cases [20]. The team then sets the goals for the above-mentioned metrics, and the DUT is intensely stressed by adding numerous tests to regression, gradually uncovering bugs and increasing the coverage until defined goals are achieved.

In the context of the described state-of-the art verification practices, we want to bring up two specific challenges. First, the described infrastructure does not guarantee that the processor is bug-free when sending the design for fabrication. The study on verification trends shows that fewer companies achieve first silicon success due to increased complexity and need costly re-spins before production. Despite the immense amount of co-simulation and achieved high coverage, 40% of the reasons that cause a re-spin are functional bugs that escape to silicon [18]. These bugs get exposed only during the silicon validation or even worse at the end-customer. The verification engineers call these bugs *outlier bugs* or *simulation resistant super bugs*. The outlier bugs exposed neither by random instruction streams nor by directed tests because the sequence of events for the bug to occur is too complicated. They can only "be exposed by exercising the design outside its normal flow or operating parameters" [5].

The second challenge is that verification requires large amounts of human effort and resources. It is reported that 53% of the whole hardware-design-cycle is spent on design verification. Out of that time, to reach the defined coverage goals, verification engineers spend 21% of their time generating tests and running them on the simulator. When bugs are uncovered, they spend 44% of their time on debugging [18].

This paper's contribution is the novel Logic Fuzzer, a technique that brings the processor's execution outside of its normal flow and increases the chances of finding outlier bugs in the simulation phase. The key in simulation-based verification is to develop the code sequence that will bring the processor into a buggy microarchitectural state that results in an inconsistent architectural state with the golden model. The Logic Fuzzer increases the microarchitectural states reached *without* the engineering effort of developing new code sequences, hence increasing the productivity of time spent on the test creation. The overall idea is to randomize states or control signals in the DUT that does not affect the correctness. For example, the re-order buffer (ROB) may assert a full or stall signal even when it is not full. It is also possible to change the branch predictor tables at any given moment or even insert the instructions in the mispredicted path. The Logic Fuzzer can change the number of cycles but does not corrupt the functionality or the program order. Our results show that atypical microarchitectural states created by Logic Fuzzer expose more bugs during the simulation phase.

Logic Fuzzer is not to be confused with the fuzzing of input stimuli [26, 30, 36]. It is the fuzzing of the actual logic. The inserted logic stirs up the execution paths while running the code and brings the processor outside its normal flow. It does not require a specialized type of code and operates independently of existing verification infrastructure.

In addition to the previous contribution, we built the state of the art co-simulation framework for RISC-V cores. We called it Dromajo [39], and it is open-source released. Dromajo is an RV64GC emulator that was designed specifically for co-simulation purposes. It can boot Linux and handle external stimuli, such as interrupts and debug requests on the fly, and be integrated into existing testbench infrastructure with minimal effort. Dromajo addresses the need for productivity and shortens the debug-verify cycle. The co-simulation framework simplifies debugging because an engineer starts the investigation at the point closest to the divergence of the model and the DUT. Our results show that the mere integration of Dromajo into the testbenches of existing open-source RISC-V cores exposed bugs and proved its effectiveness.

To the best of our knowledge, Dromajo is the only available RISC-V verification framework that supports *checkpoints*. The checkpoint is a snapshot of the processor's architectural state that is taken after executing a certain amount of instructions. Dromajo can generate such checkpoints with arbitrary RISC-V applications and resume them with the co-simulation enabled. It addresses the productivity of time spent on simulation. For example, the checkpointing capabilities of Dromajo can allow testing of lengthy running programs, such as SPEC benchmarks running on Linux in parallel. The way to achieve this is to run the program on Dromajo standalone (fast) and dump N checkpoints along the run. These checkpoints are then spawned across N different simulations and co-simulated in parallel.

To sum up, this paper has the following contributions:

- Propose and apply a novel Logic Fuzzer to find more bugs during the simulation phase.
- Contribute Dromajo, the state of the art co-simulation framework to RISC-V community. It is the artifact of the project that we used to evaluate the effectiveness of Logic Fuzzer.
- Exposure of thirteen bugs in three RISC-V cores: CVA6 [46], BlackParrot [3], and BOOM [6, 48]. Dromajo by itself found a total of nine bugs. The enhancement of Dromajo with the Logic Fuzzer increases the exposed bug count to thirteen *without* creating additional verification tests.

We demonstrate that the presented tools are capable of exposing hardware malfunctions and inconsistencies with ISA that could prevent any complex software from running correctly. The paper also provides interesting observations related to Operating Systems. Three RISC-V cores that we used for evaluation have gone through several tapeouts and claim to boot and run Linux. More than half of the bugs found were OS related. Interestingly, a "well behaved" Linux will not have excercised most of the bugs. Our results show that being able to boot and run Linux is far from saying that the core is verified.

The rest of the paper is organized as follows. Section 2 provides background concepts. Section 3 describes different types of Logic Fuzzer and implementation details. Section 4 goes over Dromajo and its main contributions. Section 5 briefly describes the evaluation methodology. We discuss the results in Section 6. Finally, we finish with the related works in Section 7 and conclude in Section 8.

2 BACKGROUND

2.1 Typical Verification Setup

In simulation-based verification, the RTL model of the microprocessor is translated into a high-level object-oriented software class [37]. The translated high-level model is then instantiated in the testbench along with the memory model. The memory gets prepopulated with the verification tests. After the simulation infrastructure is set up, we must come up with criteria to determine whether the tests that we are simulating pass or fail. The trivial option is the correctness checking by running directed tests. When a test completes execution, the final result is checked against a pre-calculated answer. The test's success or failure is determined based on the comparison of the test output and the pre-calculated answer. The directed tests are heavily used in industry, but often the purpose of these tests is to verify the basic functionality of the design or, on the contrary, to reach a well-thought-out corner case. For more than three decades, to test the design at a more rigorous level, designers both in industry and academia relied on the verification binaries that are randomly generated.

2.2 Random Instruction Generators

Random Instruction Generator, also known as Test Program Generator or Instruction Stream Generator, is a utility software that generates randomized assembly instruction streams given the set of configurations. The tests generated by the RIG sweeps a broad range of implemented functionality. It can create complex test cases that are hard to come up with for an engineer [2, 44].

Stressing the design-under-test with complete random instruction streams can be thought of as testing the system in "breadth." Complete randomness does not provide control over the generated tests. The probability of hitting a bug that is hidden "deep down" under the complex interactions among different units is very low. To close this gap, some RIGs give the ability to have control over the generated tests through test program templates. The template is an abstract description of the test and describes a set of constraints that the generator should satisfy. Hence, it is giving the ability to manage the direction and the "depth" of the generated tests [10, 41].

The next question is: how do we determine if the verification code failed or passed? Self-checking techniques are not applicable due to the random nature of the generated tests. We can solve this issue by building an infrastructure that compares the execution with the reference model.

2.3 Reference Model Comparison

The reference, or *the golden* model, is the high-level software model of a processor. The characteristics of such a model are that it is fast and uncomplicated, it does not reflect any details of the implementation and the changes to the architectural state happen in instruction-level granularity.

The reference model comparison is the verification technique that, as the name suggests, compares the execution paths of the implementation and the model. The underlying idea is simple: when we run the code on the DUT and the model, the architectural state of both must be the same at any given moment. We can implement the comparison with the reference model in different ways with different levels of complexity.

2.3.1 End-of-simulation comparison. The end-of-simulation comparison is a cheap and simplistic setup. This method compares only the architectural state once the test completes. In other words, the same code is run both on the reference model and the RTL implementation. At the end of the simulation, we dump the register file states and the memory of both and compare against each other. If any of the values do not match, the reasons are investigated. This approach's drawback is that a buggy behavior that got reflected in the architectural state can be overwritten and hidden by later correct execution. Besides, even when the mismatch is detected, another problem is that it is challenging to debug as we might be very far from the point of divergence [25].

2.3.2 Trace comparison. Another way to implement a reference model checking setup is through trace comparison. This method needs both of the models to be able to dump the execution logs. Typically, these logs contain information about program counter flow and every single register/memory writeback. The traces are then compared, and mismatches are flagged. This approach solves both of the issues mentioned above.

Nevertheless, this technique fails to work when we test an external stimulus, such as interrupts and debug requests. Due to their asynchronous nature, an external stimulus can fire completely randomly during the execution. Because in the described infrastructure, both models are running standalone and comparisons happen post factum, the single interrupt will cause execution logs to be different [27].

2.3.3 Co-simulation. To tackle the issue described in Section 2.3.2, an infrastructure that runs both models in parallel and supports communication between the models must be built. The cores simultaneously start executing the same code and pass messages to one another. In general, we must support two types of messages.

First, at the specified event, for example, when an instruction is committed, the RTL model will signal the reference to commit an instruction and compare the states of interest. The failed comparison immediately halts the execution, and the stimulus that caused it is reported. This approach simplifies debugging because an engineer starts the investigation at the point closest to the divergence.

Second, to support asynchronous interrupts, the setup must support the messaging that overwrites the emulator's execution path. When the RTL flags an interrupt, it must be able to inform the emulator to follow its execution path [1].

2.4 Formal Verification

Formal verification techniques have shown promising results [24] and are getting more popular [9, 43]. This type of verification exhaustively examines all possible execution paths. It rigorously tries to prove or disprove the correctness of a formal model of a design. Nevertheless, due to scalability issues, formal methods are applicable only on a modular level or on the designs of a moderate size and complexity [21]. To gain confidence in the system's correctness with high complexity, such as a modern microprocessor, industry still heavily relies on the dynamic verification techniques or simulation-based verification.

The simulation-based approach, on the other hand, is scalable but deals only with the finite set of execution paths. It can verify the system's correctness on the finite number of states based on the stimuli that get fed into the system. For the microprocessors, this stimuli is a verification code-base that usually consists of the following three: (1) simple directed unit-tests, (2) real applications, and (3) randomly generated instructions.

3 LOGIC FUZZER

Logic Fuzzer (LF) inserts small pieces of logic into the RTL implementation that does not break the microprocessor's functionality. We can fuzz any logic that does not affect the functionality to reach a broader range of processor microarchitectural states. Next, we discuss fuzzable logic that is typical to the modern microprocessors, but the concept can be generalized to an arbitrary digital design.

3.1 Congestors: A Case for Fuzzing

The simplest type of LF is a congestor. As shown in Figure 1, a simple example of a congestor is an or-gate inserted at the full signal of a FIFO module. The full signal gets activated even though the condition for it to become full has not been satisfied. The congestor is then randomly activated, which results in artificial backpressure. We could also locate the congestor at busy signals and ready-valid handshake signals.

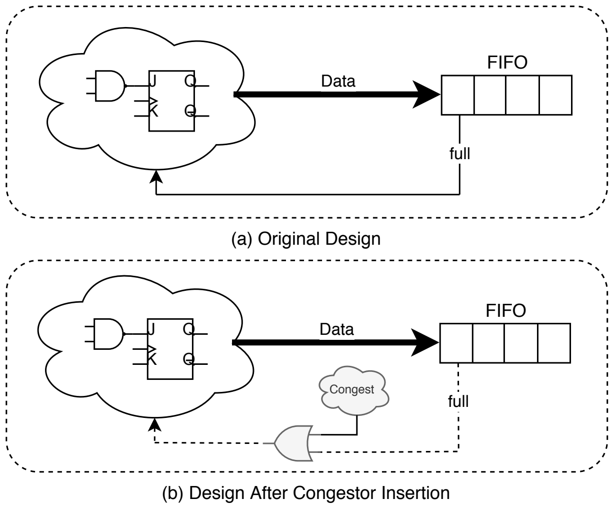

Figure 1: Congestor Logic Fuzzer placed at the FIFO's full signal

We claim that the inserted logic stirs up the execution while running the code. We can prove this by showing that after inserting the congestor logic and running the same tests, the design is experiencing different behavior, and we can observe the new activity. We can observe this by measuring the *Toggle Coverage*. The signal is said to be *toggled* if its value switched $0 \longrightarrow 1$ and $1 \longrightarrow 0$ at least once while executing the test. Toggle coverage is one of the proxy metrics that is used both in industry [38] and academia [26] to gain confidence about the correctness of the design-under-test.

For example, in the case of BOOM, we inserted a congestor at the *ready* signal of the Reorder Buffer. We then randomly pulled the *ready* signal low at the moments when the ROB was, in fact, ready. As a result, 12 additional signals toggled in the *frontend* module, 40 signals toggled in the *core* module, and 32 signals toggled in the *load-store-unit*.

For instance, according to the comments in the RTL, the signal that got activated in the load-store-unit (*execute_ignore*) "ignores the next response that comes from memory and replays it." Another example is *edge_inst* signal in the *fetchcontroller*, which gets activated when "first instruction in the bundle is PC-2."

We demonstrated that single congestor can activate logic pieces that had not been touched when running over 200 tests. Note that for this simplistic example, the usage of toggle coverage was sufficient to capture and prove that Logic Fuzzer creates additional activity. We discuss some drawbacks of the coverage metrics in 6.5.

3.2 Table Mutatators

Table mutators allow RTL memories to be mutated. For example, the tables of branch predictors can be freely fuzzed at any given moment as they must not affect the correctness of the running program. Other examples are the random invalidation of the cache or TLB entries or the value fuzzing of already invalid entries.

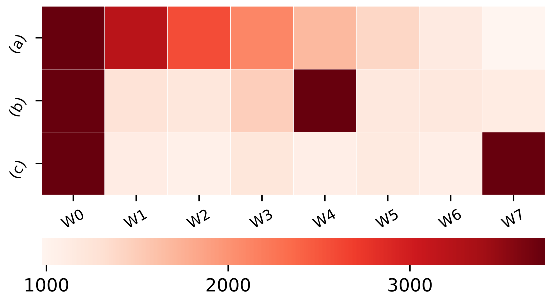

Figure 2: CVA6's L1 cache way/bank utilization without (a) tag array mutation and (b)(c) with tag array mutation

Taking an example from the application to CVA6, Figure 2 illustrates L1 cache way/bank utilization (stores only) when running over 50 random tests that were generated with Google's *riscv-dv* tool[15]. We run this set of tests three times. The first run, row (a), shows the regular run with the table mutation off. As can be seen, given the memory locations that program requests, CVA6's way selection logic gives preference to way 0. In the verification phase, an engineer may notice this fact and might decide to stress under-utilized ways. Traditionally, the engineer would have to regenerate the binaries by configuring the random instruction generator to provide the memory requests in a specific manner. The problem with this approach is that this may be a time-consuming process and will require delving into the cache replacement policies' details. Besides, the tool may not even support constraining the address generation.

The second and the third run, rows (b) and (c), illustrate that we can mutate the tag arrays and the valid bits to stir all the cache accesses to the bank of interest with the minimum amount of effort. To be specific, we edited five lines of code on the RTL side to replace tag arrays with the wrappers that access Table Mutators through DPI and the simplistic twelve-line method implementation in Table Mutator class that mutates the entries to stress the cache bank of interest.

3.3 Stressing mispredicted path

The branch predictor is a significant part of the design that has a significant effect on modern processors' performance. The advances in the branch prediction research have reached levels of accuracy exceeding 95%. However, from the verification standpoint, this means that the mispredicted path may be overlooked.

It is important to test all of the instructions in the mispredicted path as some may have side effects. Figure 3 shows the instruction coverage in the mispredicted path. The y-axis of the plot represents the number of unique RISC-V instructions that were speculatively allowed into the pipeline and eventually flushed due to the correct branch resolution. After running over 200 tests on CVA6, we see that the coverage does not reach even a 60% level. The reason is that the instruction sequence in the program exhibit nonrandom behavior and the same instructions get into the mispredicted path repeatedly. With the fuzzing, we can insert any instruction into the mispredicted path regardless of the binary. Not only can we test 100% of the instructions, but we also can reach that coverage level earlier.

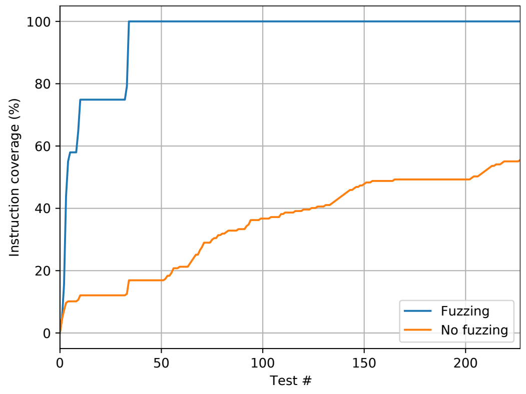

Figure 3: The coverage of instructions that were in CVA6's mispredicted path

Another interesting question to answer is: what will happen if speculative execution mechanisms will start generating instruction addresses that are atypical. Figure 4 is a scatter plot where each data point represents a Branch Target Buffer's (BTB) prediction of the PC address in a given test. The red marks are the PC predictions from when no fuzzing was enabled. We can see that the predicted addresses are within a narrow range. On the one hand, by design, the BTB is supposed to provide predictions based on the history of resolved branch target addresses. Hence, it is constrained to the address range that is encoded in the *.text* section of the *elf* file. On the other hand, the processor must be robust enough to handle non-typical cases. The data points illustrated in blue circles are the BTB's predictions when running the same tests with the fuzzing enabled. It is possible to fuzz BTB entries and provide falsely predicted addresses to a broader range or even provide random addresses at runtime. These scenarios can potentially create an iTLB page faults on the mispredicted path. The same technique can be applied to Return Address Stack.

3.4 Can the LF's states ever happen in the real world?

Logic Fuzzer could create a microarchitectural state that no program could ever reach. Nevertheless, the co-simulation failures exposed by fuzzing are potential bugs. They serve as a red flag for the engineer and must be proved or disproved to be a bug. The bugs presented in this paper were all confirmed by the designers.

3.5 Logic Fuzzer Implementation

The Logic Fuzzer as a concept can be implemented directly in the RTL code. However, to make the Logic Fuzzer integration clean,

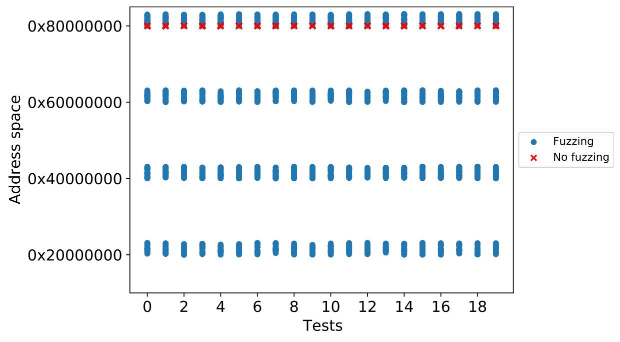

Figure 4: Instruction address ranges retrieved from Branch Target Buffer while executing random instructions.

systematic, and configurable, we embedded the LF into the existing Dromajo infrastructure. We extended the base set of APIs to provide access to the fuzzers from the RTL through the DPI calls. The fuzzers are configured by Dromajo's JSON configuration file.

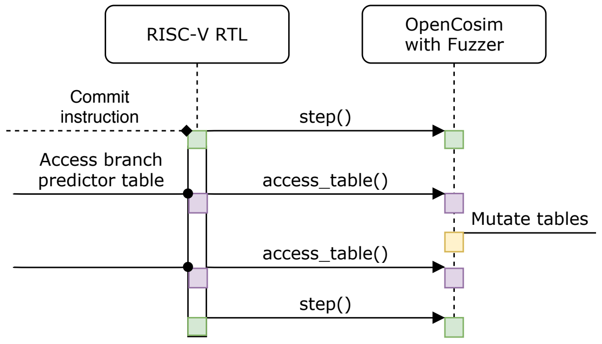

Figure 5: RTL-Fuzzer interaction flow

Figure 5 illustrates the interaction between the RTL and the fuzzers. The diagram demonstrates how the RTL implementation is accessing the table mutator of the processor's branch predictor. The fuzzer object in the Dromajo is configured to allocate the table that has the same size as the branch predictor. On the implementation side, instead of accessing the RTL memory model, it accesses the table from the fuzzer through the DPI. As the simulation is running, the tables are fuzzed randomly or with specific patterns.

For the congestors, the verification engineer first needs to identify all of the design's congestible points. The identification of these points can be consulted with the designer. After we have the list of congestible signals, the fuzzer object is configured to create the same number of congestor objects. Each congestor's period and random seeds are configured in the JSON file.

To address the productivity issue, we implemented a proof-of-concept automatic insertion of congestors into BOOM core. It allowed us to insert congestors by merely annotating the signals in RTL (one line of code per congestor). We achieved this by using *Chiffre* by Eldridge et al [12] that automatically instruments hardware systems written in Chisel [4]. Their work leverages FIRRTL compiler that allows traversal and transformation over the intermediate representation(IR) of the digital circuit with *passes* [22]. It automatically breaks the annotated signal and inserts the congestor in between. Because *Chiffre* can only work with hardware descriptions written in Chisel, this experiment was limited to BOOM core. We are currently applying the described concept using tools capable of transforming IRs generated from Verilog [42].

To insert random instructions into the mispredicted path, we make use of the table mutators. We replace the instruction cache tag and data arrays with the table mutators. The tables are written and read by the instruction cache logic through the DPI in the same manner as shown in Figure 5. We then force the Branch History Table to provide a *taken* prediction, and we force the Branch Target Buffer to provide the address with a specific tag. The fuzzer tables are then programmed to provide a random instruction stream when it sees that specific tag.

4 DROMAJO

Dromajo is an emulator designed for co-simulation with RTL processors that implement the RISC-V RV64GC instruction-set. Dromajo provides a simple set of APIs that allow for flexible integration for a variety of RTL processor implementations.

Dromajo enables executing applications, such as benchmarks running on Linux, under fast software simulation (17 MIPS). It can generate checkpoints after a given number of cycles, and resuming such checkpoints for HW/SW co-simulation. The results prove this to be a compelling way to capture bugs, especially in combination with randomized tests.

4.1 Checkpoint Definition

Dromajo checkpoints include the processor architectural state (registers, CSRs), the memory, but also reprogram interrupts (PLIC/CLINT), and performance counters such as the cycle and instructions executed. This is achieved by creating a small valid bootrom. Checkpoints in Dromajo consist of two parts: (a) memory image and (b) bootrom image. The created bootrom is a valid RISC-V program. It leverages the RISC-V debug spec that allows changing many supervisor registers. Since most RV64 CPUs support the RISC-V debug spec, the checkpoints created by Dromajo can be shared across different CPUs without requiring changes in the initialization beyond loading a different memory and bootrom.

Checkpoint based co-simulation allows to create portable stimulus and increases productivity without the need to recompile various benchmarks. A common use is to split the Linux boot sequence in several checkpoints to speed up verification. Other advantages of using checkpoints include: (a) apply concepts of phase analysis and simulation points to capture important phases in a portable format; and (b) allows a long-running program to be checkpointed and run in parallel which reduces the simulation costs.

We could leverage the concept of *Phase Analysis* [35] and checkpoint the benchmarks at the *simulation points* [34]. We then can co-simulate, for example, SPEC benchmarks in a fast manner by loading the simulation points that represent different phases of the program.

Despite the above-mentioned benefits, one disadvantage of co-simulation with checkpoints is that the branch predictor tables, caches, TLBs, and other memory elements will start the execution from the reset state. It is problematic because we lose microarchitectural states from which the bug could potentially be manifested. We believe that Logic Fuzzer's Table Mutators can partially close this gap as we can pre-populate or randomize all the tables.

4.2 Verification Flow with Dromajo

Figure 6 illustrates one possible way to implement a RISC-V core verification flow using Dromajo in five steps. The whole flow can be broken down into two main parts: checkpoint generation (Steps 1, 2, 3) and co-simulation (Steps 4, 5).

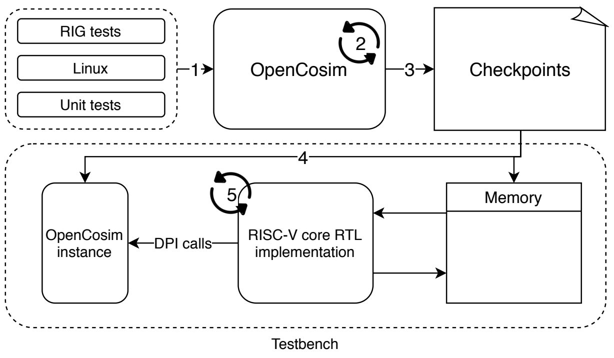

Figure 6: Verification flow with Dromajo

4.2.1 *Checkpoint Generation.*

First, Dromajo accepts an arbitrary RISC-V ELF binary (Step 1). The flow has been productive when using randomly generated tests. We then run Dromajo stand-alone (Step 2) for a certain amount of time and dump the model's whole architectural state to a checkpoint (Step 3).

4.2.2 *Step and Compare.*

As shown in Figure 6, Dromajo gets instantiated and encapsulated in the RTL as a submodule. When the simulation starts, we load the checkpoint into both models' main memories (Step 4). Dromajo instance is provided with the path to the checkpoint location. At the same time, the RTL model has to populate the main memory and initialize the content through Verilog function like *readhex*. Once the boot code completes running, both cores will have identical architectural states.

4.3 Dromajo Integration

Dromajo is compiled into a shared library. We then link this library to a simulator and interact with Dromajo through DPI calls from the Verilog. The implementation of DPI functions is trivial. Mainly, they serve the role of wrappers for Dromajo's set of API functions. Below, we explain the only three functions that we need to call to integrate Dromajo into the verification flow.

Figure 7 depicts the interactions that happen between RTL and Dromajo during the simulation. The DPI wrapper-function *cosim_init()* is invoked within an *initial* Verilog block. In turn, the *cosim_init* invokes Dromajo's initialization API function. It passes a path to the configuration file as an argument and initializes Dromajo. The

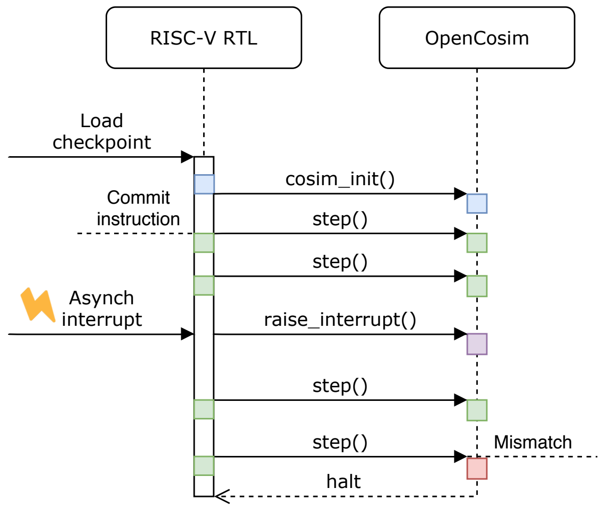

Figure 7: RTL-Dromajo interaction flow

function returns a pointer to the initialized Dromajo RISC-V reference model. The configuration file contains the path to a checkpoint and the core-specific information, such as a memory map.

The DPI wrapper-function *step()* communicates program counter, instruction and store-data to Dromajo. This function should be called whenever a valid instruction is committed (Step 5). For example, when integrating Dromajo into BOOM infrastructure, the DPI can be called by implementing simple monitor logic in the Reorder Buffer module. If the instruction at the head of the buffer is valid and ready to be committed, we call the DPI. Upon invocation, Dromajo commits one instruction on its side and conducts a comparison of communicated data. The function returns a non-zero code in case of a mismatch, and we abort the execution.

Co-simulation is a synchronous process, but interrupts are not. Hence, we need a way to log that a core took an interrupt and force the control-flow inside Dromajo to do the same. It allows us to co-simulate the interrupt trap handler routines. The DPI wrapper-function *raise_interrupt()* does that. It communicates the cause and sets the trap vector in Dromajo.

4.4 Deterministic RISC-V co-simulation

One of the prerequisites of the co-simulation is a deterministic simulation infrastructure. A common RISC-V verification infrastructure tends to use Debug Transport Module (DTM) [7] to load the test binaries to RTL and generate artificial system calls.

Interestingly, our experiments show that usage of DTM brings the core into a nondeterministic architectural state which led to false-positive co-simulation mismatches. The interaction with the host device through the memory-mapped DTM is sensitive to the characteristics and utilization of the machine running the simulator. As a result, the simulation is sometimes not deterministic. Since DTM is so common, Dromajo also supports it. However, Dromajo allows creating memory and bootram checkpoints which makes usage of DTM unnecessary. Also, avoidance of DTM usage speeds up simulation as we no longer spend time uploading the binary during the simulation. We instead prepopulate the memories before the simulation start.

By using Dromajo checkpointed infrastructure, we had a fully deterministic architectural state in the evaluated cores. Independent of this work, it may be interesting work to explore a DTM 2.0 where blocking CSR operations are implemented instead of non-blocking memory accessed to bring up binaries or model fake system calls.

5 EVALUATION METHODOLOGY

5.1 RISC-V Cores

The proposed verification tools, *i.e.*, Dromajo co-simulation enhanced with Logic Fuzzer, were evaluated on three open-source RISC-V processors. The details about each tested core are listed below and summarized in the Table 1.

5.1.1 CVA6. Previously known as Ariane and developed in ETH Zurich. The development and maintenance has been transferred to OpenHW Group. It is written in SystemVerilog. CVA6 is a 6-stage, single issue, in-order core which implements the 64-bit RISC-V instruction set. It is capable of booting Linux and was taped out in 22nm technology [46].

5.1.2 BlackParrot. Joint work of the University of Washington and Boston University. It is written in SystemVerilog. The BlackParrot is a single issue, in-order core which implements the 64-bit RISC-V instruction set. BlackParrot was written in SystemVerilog. It is capable of booting linux and 4-core configuration of BlackParrot was taped out in 12nm technology [3].

5.1.3 BOOM. Developed and maintained at UC Berkeley's *Berkeley Architecture Research* group. It is written in Chisel hardware construction language. It is a generator that can be configured to generate verilog BOOM designs with various levels of complexity. The generated cores implement the 64-bit RISC-V instruction set. We used default *MediumBoomConfig* which is a 2-wide, out-of-order core [48]. One of the more complex configurations of the BOOM was taped out in 28nm technology.

Features	CVA6	BlackParrot	BOOM
Execution	in-order	in-order	out-of-order
Issue width	1	1	2 (MedConfig)
Extenstions	RV64GC	RV64G	RV64GC
Priv. modes	M, S, U	M, S, U	M, S, U
Virt. memory	SV39	SV39	SV39

Table 1: Summary of the cores used for evaluation

5.2 Evaluation Metrics

To evaluate the proposed tools and methodology we use a simple, yet powerful metric — a precise number of bugs found. We first run a set of binaries on the base setup, i.e. testbench infrastructure with only Dromajo enabled. We then run the same set of binaries with enabled Logic Fuzzers, which expose additional bugs.

5.3 Test Binaries

We used verification binaries from two available resources: RISC-V ISA tests [14] and random instruction streams generated with Google's riscv-dv tool [15]. Table 2 summarizes the simulated test binaries that we run to get the presented results.

Core	No. of ISA tests	No. of random tests
CVA6	228	120
BlackParrot	215	150
BOOM	228	120

Table 2: Summary of the simulated tests

6 EVALUATION

6.1 Main Results

We evaluate the effectiveness of the tools on three RISC-V cores: CVA6, BlackParrot, and BOOM. We summarize our findings in Table 3. Dromajo by itself found a total of nine bugs. The enhancement of Dromajo with the Logic Fuzzer increases the exposed bug count to thirteen. Note that we did not create additional tests when enabling Logic Fuzzer. It used the same set of tests listed in Table 2 to expose additional bugs.

We demonstrated that the presented tools are capable of exposing hardware malfunctions that could prevent any complex software from running correctly. The paper also provides interesting observations related to Operating Systems (OS). Three RISC-V cores that we used for evaluation claim to boot and run Linux. More than half of the bugs found were OS related. Interestingly, a "well behaved" Linux will not have excercised most of the bugs. Our results show that being able to boot and run Linux is far from saying that the core is verified.

To gain insights on how Dromajo and Fuzzer works, we next describe the bugs found.

6.2 Bugs found by Dromajo+Logic Fuzzer

6.2.1 Bug ID B5. This bug surfaces out when we mutate the ITLB entries. The random mutation made the instruction TLB entry valid but the corresponding translation was mutated to a non-existent memory region. As a result, both Dromajo and CVA6 threw an exception and steered the execution flow to the exception handler. Dromajo halted the execution when reading an *mcause* register within the handler due to the mismatch. When trapping, Dromajo correctly sets the *mcause* value to 1 which indicates the exception cause *Instruction Access Fault.* CVA6, on the other hand, incorrectly set the value to 12 which indicates *Instruction Page Fault.* According to the designer, this is because CVA6 implementation aliases the access and page faults in the instruction front-end and treats everything as instruction page faults.

6.2.2 Bug ID B6. This bug is exposed when we create artificial backpressure at the FIFO's full signal in the cache subsystem and results in the complete hang of the system. The purpose of this FIFO is to queue memory requests that are coming from the icache. The full signal, in turn, is used to form a request logic for the arbiter, which performs arbitration between icache and dcache requests. The randomized backpressure stirs up the arbiter's states and locks the grant signal indefinitely at 0, not allowing any of the requests to go down.

6.2.3 Bug ID B11. The Black-Parrot microarchitecture defines a FIFO queue between the frontend and the backend of the core. The purpose of this FIFO is to enqueue specific commands, such as PC redirect and state reset, from the backend to the frontend. This bug is exposed when we insert the congestor at the FIFO's ready signal. We create artificial backpressure by randomly pulling this FIFO's ready signal low. When we run the tests with the inserted congestor, Dromajo catches the mismatch as BlackParrot starts committing instructions with the wrong PC. According to the designer, BlackParrot's backend cannot handle the backpressure. Because the microarchitecture has no stalling points past the decode stage, some backend commands will be lost if the queue is not ready.

6.2.4 Bug ID B12. This bug is exposed when we generate irregular addresses from BlackParrot's Branch Target Buffer (BTB). The BTB fuzzing generated the address that maps to off-chip memory. This scenario resulted in the complete freeze of the system. According to the designer, addresses that are routed to a tile must be responded to. However, BlackParrot decoded the address and routed it to a specific device on the tile, such as cfg or clint. In the case when no device matched, it hanged.

6.3 Bugs found by Dromajo

6.3.1 Bug ID B1. This bug is the result of incorrect update logic implementation of debug control status register and was exposed by Dromajo. The execution divergence occurs right after *dret* which should jump to the PC indicated by *dpc* CSR and in the privileged mode that is indicated by *prv* bits in *dcsr* CSR. Dromajo starts execution in the user-mode while CVA6 starts executes the following instruction in machine-mode. According to the designer, the confusion came from the fact that the core should update *prv* bits to the current running privileged level when entering debug mode.

6.3.2 Bug ID B2. This bug is in CVA6's integer divide unit. The unit fails to properly handle some corner case divide (*div*) and remainder (*rem*) instructions; Dromajo caught the mismatch when cores were executing division of -1/1. Dromajo committed a correct result by assigning -1 to the destination register, while CVA6 committed 0.

6.3.3 Bug ID B3. According to RISC-V ISA, *stval* CSR is written exception-specific information when the processor traps into a supervisor-mode. The ISA explicitly specifies when and which information must be written to the register. Dromajo catches a mismatch inside the exception handler when reading the value of *stval* due to incorrect setting.

6.3.4 Bug ID B4. This bug brings about similar implementation inconsistancy within the ISA specification that is described in Bug ID B3 (6.3.3). The difference here is that *mtval* control status register is written incorrect value.

6.3.5 Bug ID B7. This bug is in the BlackParrot's integer divide unit. The co-simulation failed when the BlackParrot commited *divw* insruction, which is the 32-bit integer division. The bug manifests

Bug ID	Core	Dr*	Dr+LF**	Short description	Reported	Fixed
B1	CVA6	✓		incorrect update of *prv* bits in *dcsr* register	✓	✓
B2	CVA6	✓		incorrect integer division	✓	
B3	CVA6	✓		stval CSR is written on ecall	✓	
B4	CVA6	✓		mtval CSR is written on ecall	✓	
B5	CVA6		✓	incorrect trap cause	✓	
B6	CVA6		✓	arbiter locks with gnt 0	✓	
B7	BlackParrot	✓		integer divide, incorrect handling of sign-extension	✓	✓
B8	BlackParrot	✓		no exception handling on some illegal instructions	✓	✓
B9	BlackParrot	✓		least-significant-bit not cleared on *jalr* instruction	✓	✓
B10	BlackParrot	✓		speculative long latency instructions commit	✓	✓
B11	BlackParrot		✓	core hangs on access to irregular memory region	✓	✓
B12	BlackParrot		✓	backend backpressure breaks instruction ordering	✓	✓
B13	BOOM	✓		incorrect mtval CSR value on traps	✓	✓

Table 3: Summary of the bugs exposed in three RISC-V cores. *Dr column refers to bugs found with Dromajo. **Dr+LF refers Dromajo with Logic Fuzzer

on the *remw* instruction as well. The *divw* and *remw* are signed instruction. The proper implementation should divide the lower 32-bits source registers by treating them as signed numbers, but BlackParrot's integer divide unit implementation was treating the operands as if they were unsigned.

6.3.6 Bug ID B8. This bug is in the BlackParrot's instruction decoder. The decoder did not trap an invalid instruction, but passed it down the pipeline. The machine code that BlackParrot decided not to mark as an illegal was the the binary word similar to the encoding of *jalr* instruction. The only difference was that sub-opcode *func3* was not equal to zero. The ISA defines *jalr* instruction with the subopcode of zero. The decoder had not perform any checks on *func3* bits at the time of the bug detection. Although the mismatch was detected for the case of *jalr*, in general, all instructions with invalid subopcodes should trigger an illegal instruction exception.

6.3.7 Bug ID B9. The bug is related to the calculation of the target address of the control flow instructions. The RISC-V ISA explicitly requires that the least significant bit of the calculated address by *jalr* instruction is cleared. The clearing of the bit was not implemented in the BlackParrot. Hence, Dromajo flagged the PC mismatch after the execution of *jalr* instructions. According to the designer, the confusion came from the fact the *jalr* instruction is encoded differently than *jal* and branch instructions.

6.3.8 Bug ID B10. This bug is the result of an incorrect poison bit setting. Dromajo flagged a mismatch on the load instruction. Dromajo trace analysis proved that there was only a single store-instruction to writing to the memory address. However, the following load-instruction brought a different value from what had been written. According to the designer, the values were overwritten by long latency instructions that were marked for flushing. The bug would manifest when the pipeline flushed on exceptions. Then at some point, the long latency instruction completed and allowed write-back due to the invalid poison bit.

6.3.9 Bug ID B13. This bug gets exposed when running a random instruction stream and Dromajo flags a mismatch when reading *mtval* CSR. The cores start running the binary in an M-privileged mode to execute the setup instructions, such as a series of CSR writes. Including the write to the CSR *mepc*, which gets set to *0x196*. The last instruction of the setup code is *mret*, whose execution supposed to change the privileged mode and revert the program flow to the instruction memory address pointed by *mepc*. Nevertheless, *mret* throws an exception because of the instruction page fault. According to ISA, this fault must set the *mtval* value to the address of the instruction that caused the exception. However, when we read the value of this CSR in the exception handler, they are different. The value that is set by BOOM is off by 2. According to the designer, the bug is due to the handling of compressed RISC-V instructions (RVC). Specifically, handling of exceptions on misaligned instructions appeared to be broken. The bug disappears in later commits of the BOOM.

6.4 Bug Hunting with LF and False Positives

The co-simulation mismatch is what tells if the bug was manifested or not. We start debugging only if the activity created by LF propagates to the architectural state and gets flagged by Dromajo. The process of proving or disproving if the mismatch is an actual bug is no different from regular RTL debugging. We then discuss the debugging results with the designer who confirms or rejects the validity of the finding.

Logic Fuzzer had two false bugs (not presented in the paper) – one in CVA6 and one in BOOM. Let us mention that traditional verification practices have the same issue of false positives. The fact that we find bugs in an automatic way does not qualitatively affect this issue, only quantitatively.

6.5 Toggle Coverage

The signal is said to be *toggled* if its value switched $0 \rightarrow 1$ and $1 \rightarrow 0$ at least once while executing the test. Toggle coverage is one of the proxy metrics that is used both in industry [23, 38] and academia [26] to gain confidence about the correctness of the design-under-test.

Figure 8 illustrates how the toggle coverage increases as we run the verification binaries. Logic Fuzzer increased the toggle coverage on average by 1%.

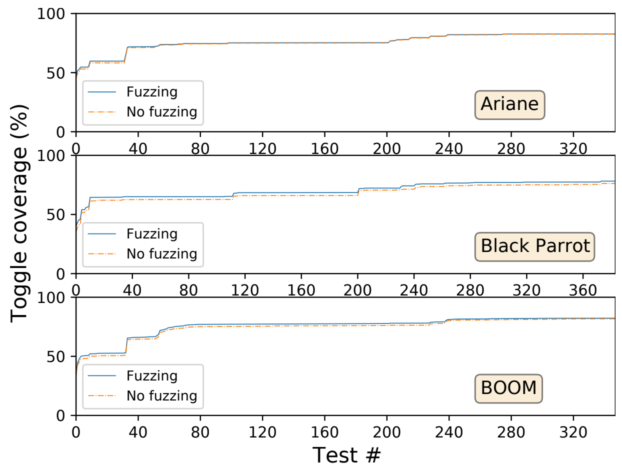

Figure 8: Coverage increase when running verification binaries

Although increased coverage is a beneficial side-effect of fuzzing, **we want to emphasize that increasing coverage in and of itself is NOT the purpose of the Logic Fuzzer. The purpose is to create an irregular execution flow, which, most of the time, is not captured by the coverage metrics**. Toggle and similar types of code coverage are not full indicators that the system is verified. They are just proxy metrics.

For example, from the bugs found by the Logic Fuzzer, only one (B12) is directly correlated to the toggle coverage. The remaining three bugs do not correlate with toggle coverage. The bugs were detected because of the randomized events created by LF. It was a combination of signals and states that even with a 100% toggle coverage may not have been detected.

7 RELATED WORK

7.1 Dromajo Related Work

Imperas Software Ltd. is a commercial company that develops virtual platforms supporting a range of ISAs, including RISC-V [29]. They claim to support step-and-compare simulation capability [28]. Imperas provide the RISC-V core *models* under the Apache 2.0. license. However the model is attached to the *simulator*, which they licensed under *OVP Fixed Platform Kit*. The difference of Dromajo with the Imperas' solution is the ability to handle checkpoints. Another difference is that the whole Dromajo is licensed under Apache 2.0.

lowRISC created a verification flow for their 32-bit RISC-V core Ibex [27]. The infrastructure that they set up executes the binary of interest both on the RTL implementation and the respective golden model. The models are completely decoupled and run the binary independently. The correctness checking happens post-completion by comparing the execution traces. In this setup, the golden model is completely oblivious to the activities happening on the RTL side. An external stimulus, such as interrupts and debug requests, will cause the traces to diverge. Therefore, this flow cannot support the instruction-by-instruction comparison in the presence of an external stimulus.

The tool proposed by Herdt et al. [19] have the co-simulation component in their flow. The novelty of their approach is that the instruction stream generator and co-simulation come in one package. They present a testbench infrastructure where instructions are generated at run-time and co-simulated endlessly. Like Ibex Core Verification flow this flow does not support the handling of the asynchronous stimuli.

Whisper [11] is the instruction set simulator developed by Western Digital. It can also be used in the co-simulation environment. The difference with Open-Cosim is that it supports only RV32, it doesn't handle interrupts, and has no support for checkpoints.

There are several general resources available for verification of a RISC-V processors. The GitHub repository developed in UC Berkeley has the set of unit tests that sweeps through the base instructions defined in ISA [14]. The RISC-V International Association has also established a Compliance Task Group [13]. Similar to [14], they only check for the basic functionality, but it is an attempt to formalize the compliance process. Currently only the RV32I ISA subset is completed. In addition, there are several open-source random instruction generators available [10, 15, 31, 33].

Finally, there is a set of works that use the term "co-simulation" in the context of heterogeneous simulation frameworks [8, 45]. In these settings, the software model is a part of a testbench and is used to drive stimuli to a design-under-test. These should not be confused with the co-simulation concept presented in this paper as we are describing it in the context of comparison with the golden model.

7.2 Logic Fuzzer Related Work

7.2.1 Input-stimuli fuzzing. Inspired by software verification techniques, several works adapted the methods for hardware verification. RFUZZ transferred the concept of American Fuzzy Lop to hardware [26]. Trippel *et al.* [40], on the contrary, explore the methods to transfer the design to the software model and apply well-established software fuzzing techniques in software domain. The technique in the PyMTL infrastructure adapted Hypothesis Testing [36]. It is property-based testing that requires to construct assertions that must always hold. The technique then tries to find the minimal possible example that breaks the assertion. Similar method presented in [30]. These techniques should not be associated with Logic Fuzzer as all of them stress the DUT externally having an "outside-in" approach. The LF proposes an "inside-out" approach, meaning the actual RTL logic is fuzzed wherever possible.

7.2.2 Fault Injection. On a surface level a parallel can be drawn between the concepts of Logic Fuzzer and Fault-Injections [12]. A fault is a physical defect or a flaw that may occur in a hardware system. As the name suggests, Fault-Injection is a simulation-based procedure when the fault is injected into the system on purpose. The simulation is then run with the fault and the behavior is observed. The idea is to prevent the system from complete failure, even in the presence of faults. The overlapping concept with the LF is the purposeful injection of logic into the system that changes the behavior.

Nevertheless, the fundamental difference is that LF makes sure that the inserted logic does not have any side effects on functionality. On the one hand, system failures detected by fault-injection are analyzed. As a result, preventive or corrective actions are proposed. On the other hand, system failures detected by Logic Fuzzer are flagged as potential functional bugs.

8 CONCLUSION AND FUTURE WORK

In this paper, we presented Logic Fuzzer, a methodology that brings the processor execution outside its normal flow by randomizing the microarchitectural state at the places that do not affect functionality. We presented several Logic Fuzzer variants and showed that it could uncover additional bugs in the simulation phase by creating atypical scenarios without the generation of additional tests. Besides, we presented Dromajo, a state of the art verification framework for RISC-V cores which addresses simulation productivity issue through checkpointing.

We illustrated the effectiveness of Dromajo by exposing nine bugs in CVA6, BlackParrot, and BOOM. The enhancement of Dromajo with Logic Fuzzer exposes additional two bugs in CVA6 and two bugs in BlackParrot. LogicFuzzer was not able to find additional bugs in BOOM. However, we demonstrated the applicability and effortless integration of the tools into existing testbench environments. Logic Fuzzer found difficult bugs. In three cases, the bugs were not directly correlated with toggle coverage, which means that the tests exercised the associated logic. The bugs were still there because a combination of events was needed to reach the bug condition. Logic Fuzzer exposes these bugs.

We are currently investigating several ways of improving the effectiveness of Logic Fuzzer. The items that we are working on include the identification and testing of other *fuzzable* logics in the microprocessor, such as reordering of outstanding memory requests and randomization of fixed priority muxes and arbiters. We are also looking into training a machine learning model to orchestrate the inserted Logic Fuzzers in the system.

We believe that Dromajo and Logic Fuzzer are great tools to help the RISC-V ecosystem have a more robust infrastructure. Besides the presented contributions, this work opens up new research opportunities and could be extended further in new areas. To mention a few, (1) Logic Fuzzer can be integrated with tools like Coppelia [47]. Coppelia uses symbolic execution techniques to generate exploits that will bring the processor into a vulnerable state. Although this work is proposed in the context of hardware security, we believe it can be extended towards functional verification. We could potentially provide Coppelia with the failing case exposed by the Logic Fuzzer and let it try to generate the code. The successful generation of the exploit will prove the presence of the real bug. (2) Integrating Logic Fuzzer in the fully elastic systems [32] could expose even more bugs than in non-elastic designs. (3) The RISC-V checkpoints created by Dromajo could also be leveraged for works that do statistical sampling for performance and power.

To conclude, we expect the developed infrastructure to enhance the quality of existing RISC-V cores, and Logic Fuzzer techniques to be applied beyond RISC-V.

9 ACKNOWLEDGMENTS

We like to thank the reviewers for their feedback on the paper. We would also like to express our deepest gratitude to the designers of the processors-under-test for their readiness to address our questions. Namely, we acknowledge Florian Zaruba from CVA6 team, Jerry Zhao and Abraham Gonzalez from BOOM team, Dan Petrisko and Farzam Gilani from Black Parrot team. This work was supported in part by grants from Google. This material is based upon work supported by, or in part by, the Army Research Laboratory and the Army Research Office under contract/grant W911NF1910466.

REFERENCES

[1] [n.d.]. [information is disclosed per double-blind peer review guidelines].

[2] W. Anderson. 1992. Logical verification of the NVAX CPU chip design. In *Proceedings 1992 IEEE International Conference on Computer Design: VLSI in Computers Processors*. 306–309.

[3] Azad, Zahra; Delshadtehrani, Leila; Zhou, Boyou; Joshi, Ajay; Gilani, Farzam; Lim, Katie; Petrisko, Daniel; Jung, Tommy; Wyse, Mark; Guarino, Tavio; Veluri, Bandhav; Wang, Yongqin; Oskin, Mark; Taylor, Michael. 2019. The BlackParrot Processor: An Open-Source Industrial-Strength RV64G Multicore Processor. (Mar 2019).

[4] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanov. 2012. Chisel: Constructing hardware in a Scala embedded language. In *DAC Design Automation Conference 2012*. 1212–1221.

[5] Brian Bailey, Harry Foster, Chirag Gandhi, Tom Anderson, Craig Shirley, Adam Sherer, Rajesh Ramanujam, Roger Sabbaugh, Vigyan Singhal, and Kevin McDermott. 2018. *When Bugs Escape)*. https://semiengineering.com/when-bugs-escape/

[6] Christopher Celio. 2018. *A Highly Productive Implementation of an Out-of-Order Processor Generator.* Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-151.html

[7] Chipyard. [n.d.]. 8.2. Communicating with the DUT. https://chipyard.readthedocs.io/en/latest/Advanced-Concepts/Chip-Communication.html

[8] Shenghsun Cho, Mrunal Patel, Han Chen, Michael Ferdman, and Peter Milder. 2018. A Full-System VM-HDL Co-Simulation Framework for Servers with PCIe-Connected FPGAs *(FPGA '18)*. Association for Computing Machinery, New York, NY, USA, 87–96. https://doi.org/10.1145/3174243.3174269

[9] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification. *Proc. ACM Program. Lang.* 1, ICFP (Aug 2017), 30 pages. https://doi.org/10.1145/3110268

[10] M. Chupilko, A. Kamkin, A. Kotsynyak, A. Protsenko, S. Smolov, and A. Tatarnikov. 2018. Test Program Generator MicroTESK for RISC-V. In *2018 19th International Workshop on Microprocessor and SOC Test and Verification (MTV)*. 6–11.

[11] Western Digital. [n.d.]. Whisper Instruction Set Simulator. https://github.com/westerndigitalcorporation/swerv-ISS

[12] Schuyler Eldridge, Alper Buyuktosunoglu, and Pradip Bose. 2018. Chiffre: A Configurable Hardware Fault Injection Framework for RISC-V Systems. In *2nd Workshop on Computer Architecture Research with RISC-V (CARRV '18)*.

[13] RISC-V Foundation. [n.d.]. https://github.com/riscv/riscv-compliance

[14] RISC-V Foundation. 2013. https://github.com/riscv/riscv-tests

[15] Google. 2019. SV/UVM based instruction generator for RISC-V processor verification. https://github.com/google/riscv-dv

[16] Mentor Harry Foster. 2019. *Part 12: The 2018 Wilson Research Group Functional Verification Study (IC/ASIC Verification Results Trends).* https://blogs.mentor.com/verificationhorizons/blog/2019/01/29/part-12-the-2018-wilson-research-group-functional-verification-study/

[17] Mentor Harry Foster. 2019. *Part 7: The 2018 Wilson Research Group Functional Verification Study (IC/ASIC Design Trends).* https://blogs.mentor.com/verificationhorizons/blog/2019/01/29/part-7-the-2018-wilson-research-group-functional-verification-study/

[18] Mentor Harry Foster. 2019. *Part 8: The 2018 Wilson Research Group Functional Verification Study (IC/ASIC Resource Trends).* https://blogs.mentor.com/verificationhorizons/blog/2019/01/29/part-8-the-2018-wilson-research-group-functional-verification-study/

[19] Vladimir Herdt, Daniel Grosse, Eyck Jentzsch, and Rolf Drechsler. 2020. Efficient Cross-Level Testing for Processor Verification: A RISC-V Case-Study. In *Forum on Specification and Design Languages (FDL)*.

[20] R. C. Ho, C. Han Yang, M. A. Horowitz, and D. L. Dill. 1995. Architecture validation for processors. In *Proceedings 22nd Annual International Symposium on Computer Architecture*. 404–413.

[21] Bo-Yuan Huang, Hongce Zhang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, and Sharad Malik. 2018. Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification. *ACM Trans. Des. Autom. Electron. Syst.* 24, 1, Article 10 (dec 2018), 24 pages. https://doi.org/10.1145/3282444

[22] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach. 2017. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 209–216. https://doi.org/10.1109/ICCAD.2017.8203780

[23] Jing-Yang Jou and Chien-Nan Liu. 1999. Coverage Analysis Techniques for HDL Design Validation. *IEEE Asia Pacific Conference on Chip Design Languages* (01 1999).

[24] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodov, Christopher Taylor, Vladimir Frolov, Erik Reeber, and Armaghan Naik. 2009. Replacing Testing with Formal Verification in Intel CoreTM i7 Processor Execution Engine Validation. In *Computer Aided Verification*, Ahmed Bouajjani and Oded Maler (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 414–429.

[25] M. Kantrowitz and L. M. Noack. 1996. I'm done simulating; now what? Verification coverage analysis and correctness checking of the DECchip 21164 Alpha microprocessor. In *33rd Design Automation Conference Proceedings, 1996*. 325–330.

[26] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. 2018. RFUZZ: coverage-directed fuzz testing of RTL on FPGAs. In *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2018, San Diego, CA, USA, November 05-08, 2018*, Iris Bahar (Ed.). ACM, 28. https://doi.org/10.1145/3240765.3240842

[27] lowRISC. 2019. Ibex Core Verification. https://github.com/lowRISC/ibex/blob/master/doc/verification.rst

[28] Imperas Software Ltd. [n.d.]. Imperas announce first reference model with UVM encapsulation for RISC-V verification. https://www.imperas.com/articles/imperas-announce-first-reference-model-with-uvm-encapsulation-for-/risc-v-verification

[29] Imperas Software Ltd. [n.d.]. OVP RISC-V Solutions. https://www.ovpworld.org/info_riscv

[30] M. Naylor and S. Moore. 2015. A generic synthesisable test bench. In *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. 128–137.

[31] OpenHWGroup. 2020. https://github.com/openhwgroup/force-riscv

[32] R. T. Possignolo, E. Ebrahimi, H. Skinner, and J. Renau. 2016. Fluid Pipelines: Elastic circuitry meets Out-of-Order execution. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*. 233–240.

[33] Anmol Sahoo. 2019. https://pypi.org/project/aapg/

[34] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically Characterizing Large Scale Program Behavior. *ACM SIGPLAN Notices* 37 (09 2002). https://doi.org/10.1145/605397.605403

[35] Timothy Sherwood, Suleyman Sair, and Brad Calder. 2003. Phase tracking and prediction. *ACM Sigarch Computer Architecture News* 31, 336– 347. https://doi.org/10.1145/871656.859657

[36] Christopher Batten Shunning Jiang, Christopher Torng. 2018. An Open-Source Python-Based Hardware Generation, Simulation, and Verification Framework. In *Proceedings of the First Workshop on Open-Source EDA Technology (WOSET18)*.

[37] Wilson Snyder. [n.d.]. Verilator. https://www.veripool.org/wiki/verilator

[38] S. Tasiran and K. Keutzer. 2001. Coverage metrics for functional validation of hardware designs. *IEEE Design Test of Computers* 18, 4 (2001), 36–45.

[39] Esperanto Technologies. 2019. Dromajo - Esperanto Technology's RISC-V Reference Model. https://github.com/chipsalliance/dromajo

[40] Timothy Trippel, Kang G. Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. 2021. Fuzzing Hardware Like Software. arXiv:2102.02308 [cs.AR]

[41] I. Wagner, V. Bertacco, and T. Austin. 2005. StressTest: an automatic approach to test generation via activity monitors. In *Proceedings. 42nd Design Automation Conference, 2005*. 783–788.

[42] Sheng-Hong Wang, Rafael Trapani Possignolo, Qian Chen, Rohan Ganpati, and Jose Renau. 2019. LGraph: A Unified Data Model and API for Productive Open-Source Hardware Design. In *Open-Source EDA Technology, Proceedings of the Second Workshop on (WOSET'19)*.

[43] Claire Wolf. 2017. SymbiYosys (sby) - Front-end for Yosys-based formal verification flows. https://github.com/YosysHQ/SymbiYosys

[44] D. A. Wood, G. A. Gibson, and R. H. Katz. 1990. Verifying a multiprocessor cache controller using random test generation. *IEEE Design Test of Computers* 7, 4 (1990), 13–25.

[45] Alberto Dassatti Xavier Ruppen, Roberto Rigamonti. 2018. Test Program Generation for Functional Verification of PowePC Processors in IBM. In *Barcelona RISC-V Workshop*.

[46] F. Zaruba and L. Benini. 2019. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 11 (Nov 2019), 2629–2640. https://doi.org/10.1109/TVLSI.2019.2926114

[47] Rui Zhang, Calvin Deutschbein, Peng Huang, and Cynthia Sturton. 2018. End-to-End Automated Exploit Generation for Validating the Security of Processor Designs. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. IEEE/ACM.

[48] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. (May 2020).