Debugging RISC-V Processors with FPGA-Accelerated RTL Simulation in the FPGA Cloud

Donggyu Kim[1], Christopher Celio[2], Sagar Karandikar[1], David Biancolin[1],
Jonathan Bachrach[1], Krste Asanovìc[1]
[1]Department of Electrical Engineering and Computer Sciences, University of California, Berkeley
{dgkim, sagark, biancolin, jrb, krste}@eecs.berkeley.edu
[2]Esperanto Technologies
christopher.celio@esperantotech.com

ABSTRACT

We present DESSERT, an FPGA-accelerated methodology for simulation-based RTL verification. The RTL design is automatically transformed and instrumented to allow deterministic simulation on the FPGA with initialization and state snapshot capture. Assert statements, which are present in RTL for error checking in software simulation, are automatically synthesized for quick hardware-based error checking. Print statements in the RTL design are also automatically transformed to generate logs from the FPGA, which are compared on the fly against a functional golden-model software simulator for more exhaustive error checking. To rapidly provide waveforms for debugging, two parallel deterministic FPGA-accelerated RTL simulations are run spaced apart in simulation time to support capture and replay of state snapshots immediately before an error. We demonstrate DESSERT with the public FPGA cloud services by catching bugs in a complex out-of-order processor hundreds of billions of cycles into SPEC2006int benchmarks running under Linux.

1 INTRODUCTION

The increasing complexity of modern hardware design makes verification challenging, and verification often dominates design costs. While formal verification approaches are increasing in capability and can be successfully employed for some blocks or some aspects of a design, and while unit-level tests can improve the confidence in individual hardware blocks, dynamic verification using simulators or emulators is usually the only feasible strategy for system-level verification. As well as verifying directed and random test stimuli, it is also important to validate the system specifications and design by running application software on the design. In addition to the large effort to create a system-level testbench, each bug found requires considerable effort to diagnose and repair.

Debugging errors found at the system-level while running realistic workloads is a notoriously difficult task. Existing approaches for system-level pre-silicon verification and debugging fall into a few categories as shown in Table 1.

Software RTL simulation with assertion detection can be an effective methodology for RTL verification and debugging, by producing waveform dumps that give full visibility into bugs. However, software RTL simulation is far too slow (up to tens of KHz) to run realistic workloads on complex hardware designs and becomes even slower when waveform dumps are enabled.

Hardware emulation engines, such as Cadence Palladium and Mentor Veloce, provide a software-like debug environment while being fast (around 1 MHz). But these custom emulation engines are extremely expensive, and can only be justified by the largest projects. Even in these projects, they remain a scarce resource that must be shared across multiple teams.

FPGA prototyping is a mainstay of pre-silicon full-system validation, as it is significantly cheaper than commercial hardware emulation engines and can be faster: single-FPGA prototypes can execute at tens to hundreds of MHz. However, FPGA prototypes provide limited visibility for signal activities, making it extremely difficult to debug any errors encountered. Moreover, many bugs are sometimes difficult to reproduce, as they may depend on the non-deterministic initial state and latencies in the host-platform, such as DRAM or network I/O. While vendors provide FPGA signal monitoring tools, such as ChipScope [20] and SignalTap [9], these require manual selection of a few signals, leading to long debug loops as the design must be re-instrumented, re-synthesized, and re-executed to change the observed signals. There has been significant research towards improving controllability and visibility in FPGA prototypes by providing GDB-like interfaces [3, 5, 10] that allow emulations to be carefully advanced, halted, and resumed, selected internal signals to be *read* and *forced*, *breakpoints* to be set at runtime, and emulation to be *rewound*. Unfortunately, like the vendor-provided tools, effective debugging is predicated on selecting the right subset of signals to be instrumented for *reads*, *forces*, and *breakpoints*.

Checkpointed FPGA prototyping removes the need to intelligently select signals to instrument [4, 7, 14, 17, 18, 21] by allowing error waveforms to be reconstructed in software RTL simulation. While this provides full visibility of the design in a region-of-interest (ROI), checkpoint intervals must be carefully chosen as frequent checkpointing of large designs can easily become a simulation bottleneck, while taking fewer snapshots lengthens the required I/O trace and the time it takes to replay the error in software simulation.

In this paper, we present DESSERT, an FPGA-accelerated methodology for effective simulation-based RTL verification and debugging with the following contributions:

- **Deterministic FPGA-accelerated RTL Simulation:** We build upon earlier work in FPGA-accelerated RTL simulators to accelerate RTL verification and debugging. We extend the Strober energy simulation framework [13], which automatically generates FPGA-accelerated RTL simulators as synchronous dataflow [15] machines to ensure *deterministic execution* given the same initial target state. We enhance the Strober automatic scan-chain insertion capability to *initialize and extract the target RTL state* for RTL debugging. The

RTL Verification Approach	Speed	Easy to Use	Deterministic	Controllability	Visibility	Cost
Software simulation	Very Slow	✓	✓	High	Full	Low
Hardware emulation engine	Fast	✓	✓	High	Full	Very High
FPGA prototype	Very Fast	✓	✗	Low	Limited	Low
Instrumented FPGA prototype	Fast	✗	✗	Moderate	Limited	High
Checkpointed FPGA prototype	Moderate	✗	✗	Low	Full	Moderate
DESSERT	Very Fast	✓	✓	High	Full in ROI	Low

Table 1: A Comparison of Contemporary Simulation Technologies for Simulation-based RTL Verification

target memory space in the off-chip DRAM is also initialized through an automatically-instrumented loadmem unit.

- **Effective Error Checking from the FPGA:** We implement custom compiler passes using FIRRTL [11] to *automatically synthesize assert and print statements* existing in RTL for *error checking from the FPGA*. Assertion synthesis provides quick hardware-based error checking with negligible simulation performance penalty. Print statement synthesis, on the other hand, provides more exhaustive software-based error checking by generating commit logs from FPGA, which are compared on the fly against a functional golden-model software simulator.
- **Low Performance Overhead on Error Detection and Replay:** Since our FPGA-accelerated RTL simulators are deterministic, we run two identical FPGA simulation instances in parallel spaced apart in simulation time to allow errors detected by the lead instance and to be replayed from an RTL snapshot captured by the trailing instance, significantly reducing state snapshotting overhead compared to periodic checkpoints. With this technique, DESSERT can provide full-visibility waveforms of a buggy design without needing to rerun the simulator and without sacrificing simulation performance.
- **System-Level Debugging with Real-World Hardware and Software:** The main contribution of this paper is to demonstrate a fast and easy-to-use methodology for system-level debugging. We demonstrate our methodology by simulating an open-source RISC-V in-order processor, Rocket [1], and an open-source RISC-V out-of-order processor, BOOM [6], to catch and fix bugs that occur hundreds of billions cycles into the SPECint2006 benchmark suite in Linux. While in this paper, we study RISC-V processors and pipe a generated commit log to a reference ISA simulator, the approach can be generalized to other RTL modules for which a golden model exists. In lieu of a golden model, inspecting synthesized assertions already present in the RTL is often a sufficient means to detect a simulation error.

2 COMPILER PASSES FOR DETERMINISTIC FPGA-ACCELERATED RTL SIMULATION

Figure 1 shows the tool flow for FPGA-accelerated RTL simulation including custom compiler passes to automatically transform the target RTL. All custom transforms are implemented as compiler passes in the FIRRTL Compiler [11]. This framework is language-agnostic: once the target designs are translated into FIRRTL from the language frontend, we can apply the compiler passes in Figure 1 regardless of their host HDLs.

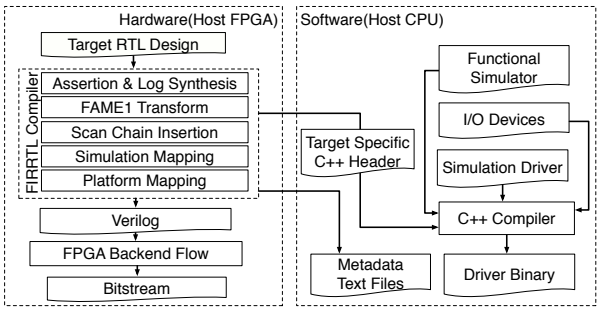

Figure 1: Toolflow for FPGA-Accelerated RTL Simulation

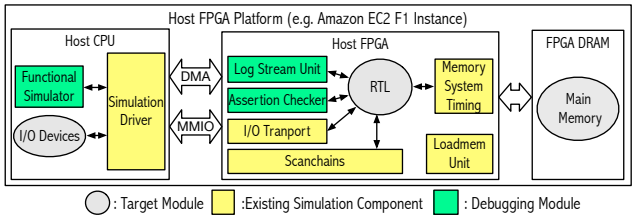

Figure 2: Mapping Simulation to the Host FPGA Platform.

This FAME1 transformation allows simulation modules to run decoupled from the host FPGA clock, which is an important optimization when all components cannot be hosted on a single FPGA. In our case studies for full-system verification, the main memory and I/O devices are mapped to the host platform memory and the software components respectively, while the RTL designs are mapped to the FPGA fabric (Figure 2).

Platform Mapping (Figure 1) links all simulation models including the FAME1-transformed RTL and abstract timing models for the main memory and I/O devices, and generates the correct interface for FPGA-software communications in various platforms. This pass also inserts the loadmem unit to initialize the memory space visible to the target design (Section 3) and helper units for bug detection including an assertion checker and a log stream unit (Section 4.3). A complete simulation system is mapped to a heterogeneous FPGA platform as shown in Figure 2. Presently, we support both Amazon EC2 F1 instances and Xilinx Zynq boards as FPGA-host platforms.

3 STATE SNAPSHOTTING AND INITIALIZATION

The Strober framework implements automatic scanchain insertion to capture RTL state snapshots for sample-based power modeling [13]. The DESSERT framework uses this technique for error replays. The scanchain implementation is extended to support target state initialization, which is necessary to initialize registers

```
1  class Count extends Module {
2    val io = IO(new Bundle {
3      val en = Input(Bool())
4      val done = Output(Bool())
5      val cntr = Output(UInt(4.W))
6    })
7    // count until 10 when `io.en' is high
8    val (cntr, done) = Counter(io.en, 10)
9    io.cntr := cntr
10   io.done := done
11
12   // assertion for software simulation
13   // `cntr' should be less than 10
14   assert(cntr < 10.U)
15
16   // printing for software simulation
17   // show the counter value when `io.en` is high
18   when(io.en) {
19     printf("count: %d\n", cntr)
20   }
21 }
```

Figure 3: Unsynthesizable Simulation Constructs in Chisel

and BRAMs that may have unexpected values after the host FPGA resets.

The off-chip DRAM should also be initialized as it is part of the target design's state. The loadmem unit (Figure 2), which is automatically added by the platform mapping (Section 2), not only loads the program to execute but also initializes the remaining target main memory space.

We also need I/O traces for error replays in software simulation. Specifically, if an RTL snapshot is to be replayed for L cycles, the inputs and the outputs for L cycles must be recorded by communication channels after the RTL snapshot is taken [13]. When the RTL snapshot is loaded in software simulation, the input traces are fed to the inputs of the target design to drive the replay, while the output traces are compared cycle by cycle against the outputs of the target design to check the correctness of the replay.

4 ERROR CHECKING FROM FPGAS

4.1 Simulation APIs in Chisel

Rocket Chip [1] and BOOM [6], the RISC-V processors featured in this case study, are written in Chisel [2], a hardware *construction* language that makes RTL design more productive via metaprogramming in a richly featured host language, Scala. Chisel makes it easy to describe *libraries* of reusable hardware generators, parameterization systems, and interconnect generators to link together a complex SoC. In Chisel version 3.0 [8], the front-end language is decoupled from the backend compiler: Chisel libraries generate FIRRTL (Flexible Intermediate Representation for RTL), while the backend compiler applies passes that gradually "lower" the FIRRTL code to Verilog [11].

Chisel, like Verilog or VHDL, provides non-synthesizable print and assert constructs for software RTL simulation. Figure 3 demonstrates their use. The module contains a counter that increments until 10 when enabled (line 8). In this example, we expect the the counter will never increment past 10: we check this with an `assert` on line 14. A `printf` (line 18-20), lets the engineer inspect the counter value without looking at the waveform. Rocket Chip and

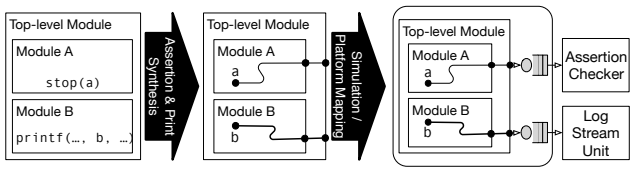

Figure 4: `stop` and `printf` Synthesis for Error Checking

BOOM use assertions extensively to check their designs. In addition to asserts, traces of important activity, like commit logs, are generated with `printf`.

4.2 Assertion and Log Synthesis

DESSERT supports two ways to detect RTL bugs from FPGAs: quick hardware-based assertion checking and more exhaustive software-based checking that compares logs against a software golden-model functional simulator. Rather than manual instrumentation, DESSERT automatically transforms assertions and logs that are already present in the source code for software RTL simulation (*Assertion and Log Synthesis* in Figure 1).

In FIRRTL, there are two constructs to support assertions and logs: `stop` and `printf` [16]. `stop` is used to halt the simulation for a certain condition, while `printf` is used to print a formatted text when its condition is met. In general, assertions in HDL (e.g. `assert` in Chisel) are expressed as `stop` with their error messages printed out by `printf`. Also, logs in HDL (e.g. `printf` in Chisel) are expressed as formatted messages in terms of RTL signal values with `printf`.

By default, `stop` and `printf` are emitted as non-synthesizable functions in (System) Verilog (e.g. `$fatal` and `$fwrite`). However, DESSERT automatically transforms `stop` and `printf` as synthesizable logic for error checking from the FPGA.

Figure 4 depicts how to automatically transform assertions and logs into synthesizable logic. Note that their conditions and arguments are logic expressions of RTL signals. Thus, *Assertion and Log Synthesis* (Figure 1) inserts the combinational logic and the signals for the conditions and the arguments of `stop` and `printf`. This pass also creates output ports and connects the signals inserted for assertions and logs to these ports so that RTL errors are detected at the boundary of the top-level module. In addition, this compiler pass emits encodings of the assertions and logs that are synthesized (e.g. the error message for each `assert` and the print format for each `printf`) into text files that are used by the software simulation driver running on the host CPU.

4.3 Handling Assertions and Logs from FPGAs

After assertions and logs are synthesized, their top-level output ports are treated in the same way as the other top-level I/Os of the target design by Simulation Mapping in Figure 1. As a result, these output ports also generate their own timing tokens, which contain the cycle-by-cycle values of the output ports, every simulation cycle (Figure 4).

The timing tokens generated by assertions and logs are crucial for *cycle-exact error checking* from FPGAs, which will deterministically occur at the same target-cycle in both software and FPGA-accelerated RTL simulations. Figure 4 also shows how these timing

tokens are handled by instrumented hardware units in FPGA, which are automatically inserted by Platform Mapping in Figure 1.

The *assertion checker* consumes timing tokens generated by assertions and inspects their values, which has no effect on simulation progress with no assertion failures. The assertion checker detects an error at cycle t if the value of the timing token at cycle t is non-zero, which means at least one assertion has fired. In this case, the checker records the target-cycle t and the assertion id inferred from the timing token's value, and then stops accepting new tokens, which will halt simulation.

In parallel, the software simulation driver infrequently polls the assertion checker through memory-mapped I/O (Figure 2), and thus cycle-exact assertion detection can be achieved with negligible loss of simulation speed. When an assertion is detected from the FPGA, the simulation driver reads the target-cycle and the assertion id from the checker and reports the assertion message along with its target-cycle using the text file generated by the Assertion and Print Synthesis pass (Section 4.2).

While the assertion checker simply drops timing tokens after inspecting them, in a log, these tokens along with their timestamps must be stored. Suppose a processor simulates at a clock rate of 50 MHz with a IPC of 0.5. If we print 64 bytes per committed instruction, this simulation would produce a commit log at 1.6 GiB/s. To manage this bandwidth, the *log stream unit* relies on inter-FPGA-CPU DMA to transfer the generated log en masse (Figure 2). Between DMA events, the log is buffered in a large BRAM FIFO. When the buffer is full, the log stream unit stops consuming timing tokens to pause simulation until the buffer is drained, which prevents loss of log entries. [1]

Once log entries are transferred from the FPGA to the buffers in the software simulation driver through DMA, they can be output on a console, piped to a file or consumed by a software golden model for exhaustive error checking.

4.4 Commit Log Comparison for Microprocessors

DESSERT is a general methodology that can be applied to any hardware designs. As such, for software-based error checking, logs generated from FPGAs are compared against a software golden model of any RTL. However, if we use DESSERT for microprocessor verification, the state of the software functional simulator must be carefully maintained to prevent divergence from the RTL implementation.

First, the physical memory and device configurations of the functional software simulator and the RTL implementation should be identical. This ensures the memory zones of Linux are the same in both implementations, resulting in the same page allocation.

Next, interrupts in both implementations must be synchronized. It is incredibly difficult to make interrupts happen simultaneously in both implementations since the functional simulator has no timing model. Instead, interrupts in the functional simulator are disabled by default. Whenever an interrupt is raised from the RTL implementation, the interrupt cause is passed along with the commit logs from the FPGA to the functional simulator. Then, the functional simulator is forced to handle the interrupt on the same target-cycle as the RTL.

In addition, microarchitecture-dependent state needs to be synchronized. Examples include performance-counter reads, atomic memory operations, memory-mapped I/Os. Performance-counter reads and atomic memory operations are easily identified by their instruction encoding while memory-mapped I/Os are identified by their memory addresses. Whenever such events happen, the destination register values of the functional simulator are updated from the FPGA's commit logs.

Some processors support out-of-order completions for long-latency instructions using a scoreboard to maintain register dependencies (e.g. the Rocket processor [1]). In this case, the destination register values may not be available even though instructions have retired. We cannot ignore these instructions due to microarchitecture-dependent state. Therefore, commit logs include the information of whether or not the scoreboard is set by each instruction. When the scoreboard is set, the destination register values are not compared immediately. Instead, the functional simulator saves the destination register value with its address. When the instruction completes in the FPGA, its destination register value as well as the register address are delivered from the FPGA to the functional simulator and compared. For microarchitecture-dependent state, the destination register value of the functional simulator is updated with the value from the FPGA.

Finally, the permission bits in TLBs are modeled in the functional simulator. This is because TLB flushes can be delayed by an OS as a performance optimization, resulting in accesses to stale page-table entries. Thus, whenever the TLBs in the FPGA are refilled, the functional simulator updates its TLB models by using the TLB tag and the permission bits of the page-table entry from the FPGA. Memory accesses in the functional simulator also go through the TLB models to match page faults between the function simulator and the FPGA.

Other forms of complex golden functional model, such as out-of-order memory systems, will require similar strategies to track cycle-level interleaving of the RTL design.

5 GANGED-SIMULATION FOR RAPID ERROR REPLAYS

To detect and replay errors efficiently, we exploit the determinism of our FPGA-accelerated simulation by running two identical simulators concurrently: a leading *master* instance, which detects the target RTL bugs, and a lagging *slave*, which checkpoints the target RTL (Figure 5).

The leading master checks for simulation errors by detecting either an assertion failure or a mismatch between the golden model and the simulator-generated log (Section 4). The master controls the advance of the slave by periodically sending it packets over TCP, each of which contains a target-cycle timestamp and an error detection bit, indicating whether or not the master has encountered an error at the timestamped target-cycle.

The slave cannot proceed until it receives a timestamped message from the master. When it receives a message with a clear error bit, it can safely advance up to the timestamped target-cycle of the message. On the other hand, when the slave receives the

[1] This may slow down simulation speed.

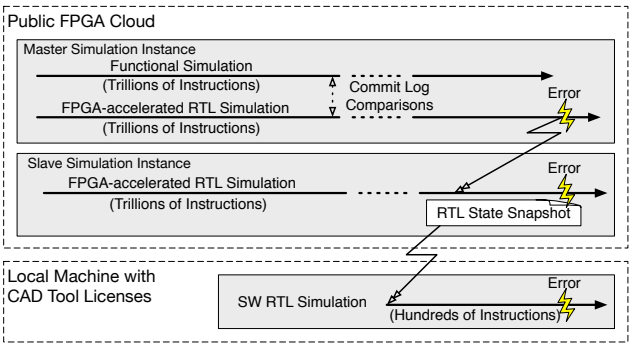

Figure 5: Ganged-Simulation For Rapid Error Relay

message with a set error bit, it advances up to the timestamped target-cycle minus L cycles to capture a L-cycle snapshot of the ROI (Section 3). Since simulations are deterministic (Section 2), the same error, which is detected by the master, also is captured by the slave at the same target-cycle.

Finally, the captured RTL state snapshot can be replayed L cycles in software RTL simulation until the same error appears, thus providing a full-visibility waveform of the target over the ROI. This waveform dramatically improves debuggability, helping RTL designers find and fix the cause of the bug.

To mitigate the monetary costs, we use FPGAs in the cloud. This provides a cheap, elastic source of very large FPGAs, without the large initial capital expense. On the other hand, commercial CAD tools are not allowed to run in the public cloud, and thus, error snapshots are copied to and replayed in the local machine with the CAD tool licenses.

6 RESULTS

We demonstrate the effectiveness of our methodology with a case study of two RISC-V processor core designs and report on the types of bugs found.

6.1 Target Designs, Golden Model, Benchmarks, and Host Platform

Target Designs: We apply DESSERT to two open-source RISC-V processors implemented with Chisel [2]: Rocket [1], a productized scalar in-order processor, and an industry-competitive, open-source out-of-order processor, BOOM-v2 [6]. Table 2 shows the processor configurations used for this study with the number of assertions and the size of log entries. Log entries are generated when instructions are committed. The processor and L1 cache represent the design-under-test (DUT) and are supplied as RTL, while the supporting L2 cache and DRAM are implemented as abstract timing models, which can be configured at runtime [12].

Software Golden Model: We employ Spike [19] as a golden model for the RISC-V ISA, which is modified for commit log comparison (Section 4.4). For software-based checking, commit logs generated by Rocket of BOOM-v2 from the FPGA are compared against Spike.

Benchmarks: We execute the SPEC2006int benchmark suite on the target processors hosted in the FPGA. All benchmarks are compiled using gcc version 6.1.0, and run on Linux kernel version 4.6.2. For each benchmark, we built a BusyBox image including all necessary files for a given benchmark within an initramfs.

Parameter	Rocket	BOOM-v2
Fetch-width	1	2
Issue-width	1	4
Issue slots	-	60
ROB size	-	80
Ld/St entries	-	16/16
Physical registers	32(int)/32(fp)	100(int)/64(fp)
Branch predictor	-	gshare: 16 KiB history
BTB entries	40	256
RAS entries	2	4
MSHR entries	2	2
L1 $ capacities	16 KiB or 32 KiB	
ITLB and DTLB reaches	128 KiB / 128 KiB	
L2 $ capacity and latency	1 MiB / 23 cycles	
DRAM capacity and latency	2 GiB / 80 cycles	
Assertions	123	601
Commit log entry width	60 B	64 B

Table 2: Parameters of the Target Processors.

Host Platform: We use Amazon F1 instances (`f1.x2large`) as simulation host platforms. An `f1.x2large` instance is equipped with Xilinx UltraScale+ VU9P and 1.5GB/s FPGA-CPU DMA.

6.2 Simulation Performance

Processor	FPGA No-Checking	FPGA Assertion	FPGA Log
Rocket	52.7 MHz	52.6 MHz	21.3 MHz
BOOM-v2	52.3 MHz	52.1 MHz	13.7 MHz

Table 3: Simulation Rates

Table 3 shows the simulation rates of FPGA-accelerated simulation with no error checking (*FPGA No-Checking*), hardware-based checking from assertion synthesis (*FPGA Assertion*), and software-based checking comparing logs from the FPGA against a golden model (*FPGA Log*).

First of all, FPGA-accelerated RTL simulation guarantees high simulation rates regardless of design complexities. In addition, hardware-based assertion checking has almost no performance overhead as the assertion checker is infrequently polled by the software driver (Section 4.3).

On the other hand, software-based checking decreases simulation rates because, in this case study, the functional simulator must be run and compared in lock step (Section 4.4). As a result, the log buffer is not quickly drained, resulting in frequent simulation stalls. Notably, software-based checking has a larger performance impact on BOOM-v2, which has higher IPC performance, and thus, generates more commit log entries per cycle. However, exhaustive software-based checking is still worthwhile as it can discover subtle bugs not found by hardware-based assertion checking (Section 6.4). We believe the simulation performance can be further improved with decoupling and speculation of functional simulation, to reduce synchronization frequency.

For comparison, the authors of the Strober framework [13] reported its simulation rate was up to 3.56 MHz, which was improved to at most 40 MHz on the Xilinx Zynq board by recent advances [12]. Note that these frameworks have no RTL debugging capabilities.

6.3 BOOM-v2 Assertion Failure Bugs Found

BOOM-v2 is a major microarchitectural update of the original BOOM processor to improve its physical realizability [6]. BOOM-v2 passes all ISA tests, random instruction tests, microbenchmark tests, and even boots Linux. However, we noticed that some of the SPECint2006 benchmarks that pass in BOOM-v1 fail in BOOM-v2. Therefore, we used DESSERT to debug BOOM-v2.

Benchmark	Assertion Failure	Cycle (B)	Simulation Time (mins)
483.xalancbmk.test	Invalid writeback in ROB	1.9	3.4
464.h264ref.test	Pipeline hung	3.2	3.8
471.omnetpp.test	Pipeline hung	3.3	3.9
445.gobmk.test	Invalid writeback in ROB	14.9	9.0
471.omnetpp.ref	Pipeline hung	62.6	22.2
401.bzip2.ref	Wrong JAL target	473.7	164.6

Table 4: Assertion Triggers from BOOM-v2 running the SPEC2006int Benchmark Suite.

Table 4 shows assertions caught from BOOM-v2 when running the SPECint2006 benchmarks. Note that assertion messages are shown in FPGA-accelerated RTL simulation when these assertions are triggered. In addition, RTL state snapshots are taken before the assertions are triggered (Section 5) and replayed in software RTL simulation for full visibility of the internal signals.

With the waveform from the 1024-cycle error replay, we quickly tracked down the cause of the *invalid writeback in ROB* assertion to a buggy interaction between back-pressure queuing and branch misspeculation that did not correctly kill instructions moving data from the integer register file to the floating-point register file. In general, the *pipeline hung* assertion is caused by pipeline resource scarcities for various reasons, which are not found in the 1024-cycle window, suggesting assertions describing more specific properties be necessary. Also, the waveform from the 1024-cycle error replay reveals the *wrong JAL target* assertion, which is triggered at almost a half trillion target cycles, was caused by incorrectly handled signed arithmetic in computing jump target addresses, which is latent until the processor touches instructions allocated in a high address memory region.

We caught all these assertion triggers and obtained full visibility within 3 hours using two Amazon EC2 F1 instances. Therefore, the total cost to catch and replay these errors is roughly $2 (compilation) $+2 \times \$1.56$ (simulation) = $5.12 with spot instances, which is extremely economical compared to commercial emulation tools.

6.4 Boom-v2 Commit Log Bugs found

Software-based checking comparing logs from an FPGA against a software golden model can discover subtle bugs that may not immediately affect the results of applications. We verify Linux boot in Rocket and BOOM against the software golden model using commit logs from the FPGA (Section 4.4). Linux boot in Rocket is successfully verified against the golden model. [2] However, Linux boot in BOOM-v2 fails with the following message:

```
Instruction mismatch at cycle: 669432906
    PRIV        PC                INST                REG
Last: 0 0x0000000000069ce0 (0x00100793) x15 0x0000000000000001
SW  : 0 0x0000000000069ce4 (0x1404272f) x14 0x0000000000000000
FPGA: 1 0xffffffff80422a9c (0x14011173) x 2 0xffffffffcc54000
```

This shows BOOM jumps into Linux's exception handler (PC = `0xffffffff80422a9c`) while executing `lr.w a4, zero, (s0)` (`0x1404272f`). The waveform from the 1024-cycle replay shows BOOM incorrectly triggers a store access fault for load-reserved instructions. After fixing this bug, Linux boot in BOOM-v2 fully matches against the golden model. This bug was found in less than three minutes including target memory initialization, but would have taken a month using VCS.

Commit log comparisons are also helpful to catch bugs that are not easily discovered by assertions. For example, `403.gcc.test` fails in BOOM without assertion triggers. However, from commit logs, the following mismatch is found:

```
Instruction mismatch at cycle: 2909587019
    PRIV        PC                INST                REG
Last: 0 0x00000000001d15fc (0x14d76e63)
SW  : 0 0x00000000001d1600 (0x03079793) x15 0x0000000000000000
FPGA: 0 0x00000000001d1600 (0x01813483) x 9 0x00000000004322e8
```

Note that this bug is found at 2.9 billion cycles in just *6 minutes*; Verilator would have taken nearly *three weeks* to reach this bug.

The commit log shows BOOM fetching the wrong instruction at PC = `0x1d600`. The waveform from the 1024-cycle replay shows that BOOM's fetch buffer is unable to accept more instructions and applies back-pressure to the instruction cache, which experiences a cache miss at the same time. Once the cache miss is resolved, the wrong instruction is returned from the instruction cache. BOOM-v2 shares the frontend and the instruction cache with RocketChip, and we used an old version of RocketChip [3] on which the current version of BOOM-v2 [4] is based. The frontend and the instruction cache in the current version of RocketChip has since been completely rewritten. We will verify BOOM-v2 again with a newer RocketChip code base in the future.

7 CONCLUSION

By automatically transforming target RTL into an instrumented FPGA-accelerated simulator and connecting the FPGA simulator to a tracking functional golden model for checking, we can rapidly find and diagnose bugs that only manifest after hundreds of billions of target clock cycles, with little developer effort and at extremely low cost, by taking advantage of cloud-hosted FPGA platforms.

ACKNOWLEDGEMENT

Research partially funded by DARPA Award Number HR0011-12-2-0016, RISE Lab sponsor Amazon Web Services, and ADEPT/ASPIRE Lab industrial sponsors and affiliates Intel, HP, Huawei, NVIDIA, and SK Hynix. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and do not necessarily reflect the position or the policy of the sponsors.

[2] We could not easily match floating-point loads due to what was a legally valid ambiguity due to microarchitectural implementation differences between Rocket Chip and the golden model (Spike). Newer versions of the RISC-V ISA close this specification ambiguity.

[3] Commit Hash: 8c8d2af7141102adf8ccc65b929e740ce7ce189, Date: Feb 9th, 2017

[4] Commit Hash: 70b94eefe6658a1444ca420ab86953c25665dae8, Date: Sep 12th, 2017

REFERENCES

[1] Krste Asanović et al. 2015. *The Rocket Chip Generator.* Technical Report UCB/EECS-2016-17.

[2] Jonathan Bachrach et al. 2012. Chisel: constructing hardware in a scala embedded language. In *DAC*.

[3] Somnath Banerjee and Tushar Gupta. 2012. Efficient online RTL debugging methodology for logic emulation systems. In *VLSI*.

[4] Somnath Banerjee and Tushar Gupta. 2012. Fast and scalable hybrid functional verification and debug with dynamically reconfigurable co-simulation. In *ICCAD*.

[5] Kevin Camera and Robert W. Brodersen. 2008. An integrated debugging environment for FPGA computing platforms. In *FPL*.

[6] Christopher Celio, Pi-Feng Chiu, Borivoje Nikolić, David A. Patterson, and Krste Asanović. 2017. BOOMv2: an open-source out-of-order RISC-V core. In *First Workshop on Computer Architecture Research with RISC-V (CARRV)*.

[7] Chin-Lung Chuang, Wei-Hsiang Cheng, Chien-Nan Jimmy C.-N.J. Liu, Dong-Jung Lu, and Chien-Nan Jimmy C.-N.J. Liu. 2007. Hybrid Approach to Faster Functional Verification with Full Visibility. *IEEE Design & Test of Computers* 24, 2 (2007), 154–162.

[8] FreeChips Project. 2017. Chisel 3 wiki. (2017). https://github.com/freechipsproject/chisel3/wiki

[9] Intel. 2017. SignalTap II Logic Analyzer: Introduction & Getting Started (ODSW1164). (2017). https://www.altera.com/support/training/course/odsw1164.html

[10] Yousef S. Iskander, Cameron D. Patterson, and Stephen D. Craven. 2011. Improved abstractions and turnaround time for FPGA design validation and debug. In *FPL*.

[11] Adam Izraelevitz et al. 2017. Hardware Reusability is FIRRTL Ground: Hardware Construction Languages, Compiler Frameworks, and Transformations. In *ICCAD*.

[12] Donggyu Kim, Christopher Celio, David Biancolin, Jonathan Bachrach, and Krste Asanović. 2017. Evaluation of RISC-V RTL with FPGA-Accelerated Simulation. In *First Workshop on Computer Architecture Research with RISC-V*.

[13] Donggyu Kim, Adam Izraelevitz, Christopher Celio, Hokeun Kim, Brian Zimmer, Yunsup Lee, Jonathan Bachrach, and Krste Asanović. 2016. Strober : Fast and Accurate Sample-Based Energy Simulation for Arbitrary RTL. In *ISCA*.

[14] Dirk Koch, Christian Haubelt, and Jürgen Teich. 2007. Efficient hardware checkpointing: concepts, overhead analysis, and implementation. In *FPGA*.

[15] E.A. Lee and D.G. Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (1987), 1235–1245.

[16] Patrick S Li, Adam M Izraelevitz, and Jonathan Bachrach. 2016. *Specification for the FIRRTL Language.* Technical Report UCB/EECS-2016-9.

[17] J. Marantz. 1998. Enhanced visibility and performance in functional verification by reconstruction. In *DAC*.

[18] Andrew G. Schmidt, Bin Huang, Ron Sass, and Matthew French. 2011. Checkpoint/restart and beyond: Resilient high performance computing with FPGAs. In *FCCM*. https://doi.org/10.1109/FCCM.2011.22

[19] Andrew Waterman and Yunsup Lee. 2011. Spike, a RISC-V ISA Simulator. (2011). https://github.com/riscv/riscv-isa-sim

[20] Xilinx. 2017. ChipScope Pro and the Serial I/O Toolkit. (2017). https://www.xilinx.com/products/design-tools/chipscopepro.html

[21] Zan Yang, Byeong Min, and Gwan Choi. 2000. Si-emulation: system verification using simulation and emulation. In *International Test Conference*.