DESSERT: Debugging RTL Effectively with State Snapshotting for Error Replays across Trillions of cycles

Donggyu Kim[1], Christopher Celio[2], Sagar Karandikar[1], David Biancolin[1], Jonathan Bachrach[1], Krste Asanović[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, running on public-cloud FPGAs at extremely low cost, by catching bugs in a complex out-of-order processor hundreds of billions of cycles into SPEC2006int benchmarks running under Linux.**

***Keywords*-RTL debugging; hardware verification; simulation-based verification; FPGA-accelerated simulation**

I. 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 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 I.

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 and SignalTap, 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 [1], [2], [3] 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], [5], [6] 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:

- We implement custom compiler passes using FIRRTL [7] 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 a negligible simulation performance penalty. Print-statement synthesis, on the other hand, provides more exhaustive software-based error checking by generating commit logs from the FPGA, which are compared on the fly against a functional golden-model software simulator.
- 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 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.
- We demonstrate a fast and easy-to-use methodology for system-

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 I: A Comparison of Contemporary Simulation Technologies for Execution-driven RTL Verification

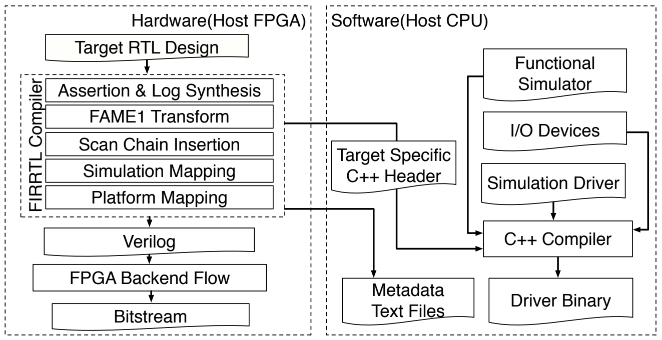

Fig. 1: Toolflow for FPGA-Accelerated RTL Simulation

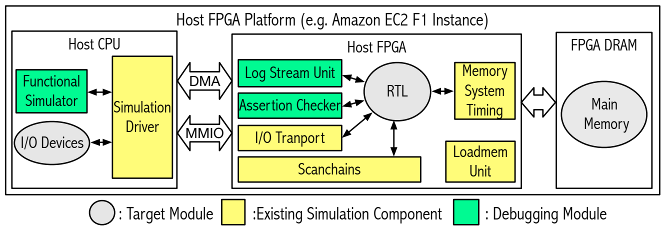

Fig. 2: Mapping Simulation to the Host FPGA Platform.

level debugging. We demonstrate our methodology by simulating an open-source RISC-V in-order processor, Rocket [8], and an open-source RISC-V out-of-order processor, BOOM [9], to catch and fix bugs that occur hundreds of billions cycles into the SPECint2006 benchmark suite running under Linux. While in this paper we study RISC-V processors and pipe a generated commit log to a reference ISA simulator, our approach can be generalized to other RTL designs 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.

II. 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 [7]. This framework is language-agnostic: once the target design is translated into FIRRTL from its language frontend, we can apply the compiler passes in Figure 1 regardless of the design's host HDL.

The *FAME1 transform* allows simulation modules to run decoupled from the host FPGA clock to stall simulation when necessary. *Simulation Mapping* inserts communication channels, wrapping a simulation module to buffer timing tokens from other simulation modules. Most importantly, a token-based simulator generated by these compiler passes is an instance of synchronous dataflow (SDF) [10], which ensures *deterministic execution on the FPGA with the same initial state*.

Platform Mapping 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*, used to initialize the memory space visible to the target design (Section III), and helper units for bug detection, such as an assertion checker and a log stream unit (Section IV-B). 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.

III. State Snapshotting and Initialization

The Strober framework implements automatic scanchain insertion to capture RTL state snapshots for sample-based power modeling [11]. 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 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 II), 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 [11]. 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.

IV. Error Checking from FPGAs

A. Assertion and Log Synthesis

DESSERT supports two ways to detect RTL bugs: 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`. `stop` is used to halt the simulation for a certain condition, while `printf` is used to print a formatted string 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` into synthesizable logic for error checking from the FPGA.

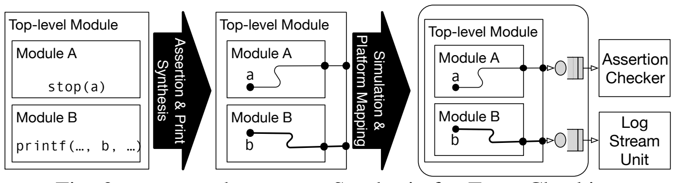

Fig. 3: `stop` and `printf` Synthesis for Error Checking

Figure 3 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.

B. 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 3).

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 3 also shows how these timing tokens are handled by instrumented hardware units in the 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 IV-A).

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 an 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.

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.

C. 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, and 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 [8]). 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.

V. 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

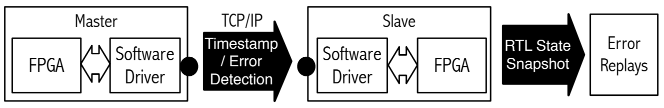

Fig. 4: Ganged-Simulation For Rapid Error Relay

RTL bugs, and a lagging *slave*, which checkpoints the target RTL state (Figure 4).

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 IV). 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 message with a set error bit, it advances up to the timestamped target cycle minus L cycles to capture an L-cycle snapshot of the ROI (Section III). Since simulations are deterministic (Section II), 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.

VI. 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.

A. Target Designs, Golden Model, Benchmarks, and Host Platform

Target Designs: We apply DESSERT to two open-source RISC-V processors implemented with Chisel [12]: Rocket [8], a productized scalar in-order processor, and an industry-competitive, open-source out-of-order processor, BOOM-v2 [9]. Table II 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 [13].

Software Golden Model: We employ Spike [14] as a golden model for the RISC-V ISA, which is modified for commit log comparison (Section IV-C). For software-based checking, commit logs generated by Rocket or BOOM-v2 from the FPGA are compared against Spike.

Benchmarks: We execute the SPEC2006int benchmark suite on the target processors hosted on 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.

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.

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 II: Parameters of the Target Processors.

B. FPGA Quality Of Results

We compiled bitstreams using Vivado 2017.1 targeting the Xilinx UltraScale+ VU9P parts present in Amazon EC2 F1 instances. Pure FPGA mappings for both designs close timing at 62.5 MHz, which is bounded by the unretimed double-precision FMA in both cores. The compile time is about 2 hours for Rocket and 4 hours for BOOM on `c4.8xlarge` (about \$1 and \$2 with spot instances, respectively).

Processor	Resource	Prototype	FAME1	Debug	Scan	All
Rocket	Logic LUTs	18.0%	18.4%	18.5%	24.6%	24.7%
	Registers	10.8%	10.8%	10.9%	13.6%	13.7%
	BRAMs	18.1%	19.6%	24.9%	21.2%	26.6%
BOOM-v2	Logic LUTs	28.0%	28.4%	30.7%	51.5%	52.1%
	Registers	12.9%	12.8%	13.4%	22.4%	22.5%
	BRAMs	19.4%	20.9%	27.4%	22.6%	30.1%

TABLE III: FPGA Utilization vs Instrumentation Level

Table III shows the total utilization of the VU9P after place and route, with varying levels of instrumentation enabled:

- *Prototype*: just the processor without transformations
- *FAME1*: FAME1 simulator for deterministic simulation (Section II)
- *Debug*: FAME1 simulator with assertion and print synthesis (Section IV)
- *Scan*: FAME1 simulator with scan chains insertion (Section III)
- *All*: FAME1 simulator with all transforms and instrumentation

LUTRAMs, DSP48s, and URAMs are omitted as they are lightly used (<1%, <5% and 0%).

The FAME1 transform has marginal overhead over the prototype due to FPGA tool optimizations. The debug instrumentation uses slightly more LUTs for assertion synthesis and more BRAMs for log buffers. As expected, the scanchain instrumentation has large overhead on both LUTs and Registers. However, the DESSERT framework can be extended to adopt more resource-efficient checkpoint implementation as discussed by Koch et al [6].

C. 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 IV: Simulation Rates

Table IV shows the simulation rates of a single instance 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 IV-B).

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 IV-C). 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 greater IPCs, 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 VI-E). We believe the simulation performance can be further improved with decoupling and speculation of functional simulation, to reduce synchronization frequency.

D. BOOM-v2 Assertion Failure Bugs Found

BOOM-v2 is a major microarchitectural update of the original BOOM processor to improve its physical realizability [9]. BOOM-v2 passes all ISA tests, random instruction tests, microbenchmark tests, and boots Linux. However, we noticed that some of the SPECint2006 benchmarks that passed in BOOM-v1 failed 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 V: Assertion Triggers from BOOM-v2 Running the SPEC2006int Benchmark Suite.

Table V shows assertions caught from BOOM-v2 when running the SPECint2006 benchmarks. Note that assertion messages were shown in FPGA-accelerated RTL simulation when these assertions were triggered. In addition, RTL state snapshots were taken before the assertions were triggered (Section V) 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 was caused by pipeline resource scarcities for various reasons, which were not found in the 1024-cycle window, suggesting assertions describing more specific properties be necessary. Also, the waveform from the 1024-cycle error replay revealed the *wrong JAL target* assertion, which was 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.

E. 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 verified Linux boot in Rocket and BOOM against the software golden model using commit logs from the FPGA (Section IV-C). Linux boot in Rocket was successfully verified against the golden model. However, Linux boot in BOOM-v2 failed 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 jumped into Linux's exception handler (PC = `0xffffffff80422a9c`) while executing `lr.w a4, zero, (s0) (0x1404272f)`. The waveform from the 1024-cycle replay showed BOOM incorrectly triggered a store access fault for load-reserved instructions. After fixing this bug, Linux boot in BOOM-v2 fully matched 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.

VII. 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, Google, Huawei, NVIDIA, and SK Hynix. Donggyu Kim is supported in part by the Kwanjeong Educational Foundation. 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.

References

[1] K. Camera and R. W. Brodersen, "An integrated debugging environment for FPGA computing platforms," in *FPL*, 2008.
[2] Y. S. Iskander *et al.*, "Improved abstractions and turnaround time for FPGA design validation and debug," in *FPL*, 2011.
[3] S. Banerjee and T. Gupta, "Efficient online RTL debugging methodology for logic emulation systems," in *VLSI*, 2012.
[4] Z. Yang *et al.*, "Si-emulation: system verification using simulation and emulation," in *International Test Conference*, 2000.
[5] C.-L. Chuang *et al.*, "Hybrid Approach to Faster Functional Verification with Full Visibility," *IEEE Design & Test of Computers*, vol. 24, no. 2, pp. 154–162, 2007.
[6] D. Koch *et al.*, "Efficient hardware checkpointing: concepts, overhead analysis, and implementation," in *FPGA*, 2007.
[7] A. Izraelevitz *et al.*, "Hardware Reusability is FIRRTL Ground: Hardware Construction Languages, Compiler Frameworks, and Transformations," in *ICCAD*, 2017.
[8] K. Asanović *et al.*, "The Rocket Chip Generator," Tech. Rep. UCB/EECS-2016-17, 2015.
[9] C. Celio *et al.*, "BOOMv2: an open-source out-of-order RISC-V core," in *First Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2017.
[10] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.
[11] D. Kim *et al.*, "Strober : Fast and Accurate Sample-Based Energy Simulation for Arbitrary RTL," in *ISCA*, 2016.
[12] J. Bachrach *et al.*, "Chisel: constructing hardware in a scala embedded language," in *DAC*, 2012.
[13] D. Kim, *et al.*, "Evaluation of RISC-V RTL with FPGA-Accelerated Simulation," in *First Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2017.
[14] A. Waterman and Y. Lee, "Spike, a RISC-V ISA Simulator," 2011. [Online]. Available: https://github.com/riscv/riscv-isa-sim