Debugging in the Brave New World of Reconfigurable Hardware

Jiacheng Ma
University of Michigan

Gefei Zuo
University of Michigan

Kevin Loughlin
University of Michigan

Haoyang Zhang
University of Michigan

Andrew Quinn
University of California, Santa Cruz

Baris Kasikci
University of Michigan

ABSTRACT

Software and hardware development cycles have traditionally been quite distinct. Software allows post-deployment patches, which leads to a rapid development cycle. In contrast, hardware bugs that are found after fabrication are extremely costly to fix (and sometimes even unfixable), so the traditional hardware development cycle involves massive investment in extensive simulation and formal verification. Reconfigurable hardware, such as a Field Programmable Gate Array (FPGA), promises to propel hardware development towards an agile software-like development approach, since it enables a hardware developer to patch bugs that are detected during on-chip testing or in production. Unfortunately, FPGA programmers lack bug localization tools amenable to this rapid development cycle, since past tools mainly find bugs via simulation and verification. To develop hardware bug localization tools for a rapid development cycle, a thorough understanding of the symptoms, root causes, and fixes of hardware bugs is needed.

In this paper, we first study bugs in existing FPGA designs and produce a testbed of reliably-reproducible bugs. We classify the bugs according to their intrinsic properties, symptoms, and root causes. We demonstrate that many hardware bugs are comparable to software bug counterparts, and would benefit from similar techniques for bug diagnosis and repair. Based upon our findings, we build a novel collection of hybrid static/dynamic program analysis and monitoring tools for debugging FPGA designs, showing that our tools enable a software-like development cycle by effectively reducing developers' manual efforts for bug localization.

CCS CONCEPTS

• **Hardware → Reconfigurable logic and FPGAs**; • **Software and its engineering → Software testing and debugging**.

KEYWORDS

FPGA, Reconfigurable Hardware, Bug Study, Debugging

ACM Reference Format:
Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Haoyang Zhang, Andrew Quinn, and Baris Kasikci. 2022. Debugging in the Brave New World of Reconfigurable Hardware. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3503222.3507701

1 INTRODUCTION

Field Programmable Gate Arrays (FPGAs) are increasingly prominent in modern heterogeneous computer systems. Specialized hardware designs provide unprecedented efficiency in domains such as machine learning [74, 83, 101, 102, 122, 127, 128], compression [92, 125], database operations [88, 96, 104], graph processing [36, 47, 112, 129], networking [41, 52, 111], and storage virtualization [78]. To realize the benefits of FPGAs, systems researchers have built operating systems [53, 73, 77, 106], virtualization support [42, 46, 80, 85, 113, 120, 123, 124], just-in-time compilers [97], and high-level synthesis tools [43, 44, 61, 116, 117]. The proliferation and benefits of FPGAs have even prompted major cloud vendors to provide FPGA instances on their platforms [31, 33].

Compared to traditional hardware development, FPGA development has many similarities to software development. Since post-fabrication bugs are extremely costly to fix, traditional hardware development invests massive resources into simulation-based testing and formal verification to eradicate bugs *before* silicon fabrication. In contrast, reconfigurability allows a developer to patch hardware bugs in an FPGA, even those caught during on-FPGA testing or in production. As a result, FPGA developers are moving towards an agile development approach that accelerates time to market by relaxing cumbersome verification in favor of lightweight simulation and on-FPGA testing. For example, Microsoft has adopted a software-like methodology for FPGA development, in which they perform relatively small amounts of verification compared to traditional hardware [52].

Unfortunately, relaxed verification leads to more bugs in FPGA designs, with most FPGA projects experiencing bugs that escape testing and end up in production [54]. Alas, while there are many hardware tools that help developers *find* bugs using simulation-based testing and verification [30, 63, 79, 81, 105, 110, 114, 115, 126], very few hardware debugging tools help a developer *localize the root cause* of a bug. Existing fault localization tools only apply to specific protocols and algorithms [28, 86]. Other tools, such as checkpointing [16, 37, 38, 75, 103] and tracing [55–57, 62, 80, 97, 119], can be used to localize the cause of a hardware failure, but require substantial manual effort to do so. Finally, existing software fault localization techniques, such as data-race detectors [99] and undefined memory use detectors [98], cannot be immediately applied to hardware programming models. Consequently, debugging an FPGA design today is a highly manual process that either involves inspecting a massive waveform (i.e., a trace of the state of the circuit over time) or iterative rounds of synthesis in which a developer selects and analyzes key data signals. Unsurprisingly, a majority of FPGA developers in a recent study indicate a need for better debugging tools [23].

Reaping the full benefits of rapid FPGA development will require constructing FPGA debugging tools that help localize the root cause of a hardware fault—similar to the rich set of tools available to software developers. Towards this goal, we first study bugs in open-source FPGA designs. We then introduce a novel root-cause-based classification of the bugs we study inspired by a prior bug taxonomy [82] and document the intrinsic properties and symptoms of these bugs. We augment our study with a testbed in which each hardware bug is reliably reproducible. We demonstrate that each class of hardware bugs mirrors a counterpart class of software bugs and would benefit from similar techniques for bug diagnosis and repair.

Guided by the intrinsic properties and symptoms of bugs in FPGA designs, we build a collection of hybrid static/dynamic program analysis and monitoring tools to help developers of reconfigurable hardware systems follow a software-like development and debugging process. Because hardware bugs may be detected during simulation or when executing on an FPGA, our tools are designed to operate in either scenario. Thus, we consider the effects of our debugging logic on real circuit synthesis and behavior, as opposed to only accounting for a simulator environment where resource and timing constraints are far less stringent. At a high level, our tools allow selectively recording and analyzing targeted execution information using limited on-FPGA storage, and consist of the following:

1- SignalCat unifies hardware debugging during simulation and when deployed on an FPGA by providing a single interface for tracing state in a hardware design. The tool converts "`printf`"-like statements embedded in a hardware description into logic that records the arguments of these statements in a hardware deployment or during simulation. After an execution, SignalCat reconstructs a log containing the output of the `printf` statements.

2- FSM Monitor helps a developer identify and track finite state machines (FSMs), which are a widespread component in a hardware design. It uses SignalCat to support both simulation and on-FPGA scenarios.

3- Dependency Monitor enables a developer to trace the provenance of the value of a variable in their hardware design. The tool identifies the dependency chain of each developer-specified variable (i.e., the registers upon which the variable depends), and tracks all updates made to these variables during a simulation or on-FPGA execution using SignalCat.

4- Statistics Monitor helps a developer identify anomalous behavior by recording statistics about various execution events, such as the number of times that an interrupt is triggered or the number of packets that arrive in a communication channel. Developers specify an event of interest; Statistics Monitor instruments the hardware design with new logic that uses SignalCat to track statistics during simulation or on-FPGA scenarios.

5- LossCheck helps a developer localize the root cause of data loss (e.g., an unintended packet drop). A developer who suspects data loss in their design uses LossCheck to check for—and potentially identify the source of—data loss between a specified source (e.g., an input to a hardware module) and sink (e.g., an output). The tool instruments the hardware design with new logic that monitors all data propagation paths between the source and sink by using SignalCat.

We show how a developer can use the aforementioned tools—either individually or in various combinations—to debug the bugs in our study. In particular, we show that our tools help diagnose the cause of each bug in our study by automatically generating and executing dozens to thousands of lines of analysis code, which the developer would otherwise need to write. Additionally, we evaluate the resource overhead of our debugging tools and demonstrate that they are feasible for production use. Among the 20 bugs we evaluated, 18 cases maintain the design's original target frequency after debugging instrumentation; all cases incur at most linear resource overheads with increased recording buffer sizes.

Overall, we make the following contributions:

- We provide the first study of bugs in open-source FPGA designs, a root-caused-based bug classification, and a description of typical bug symptoms to guide developers in their debugging efforts.
- We design a collection of hybrid static/dynamic analyses that developers can use in simulation and real hardware deployments to debug FPGA designs.
- We develop an open-source testbed [1] that includes reproducible hardware bugs and our tools to facilitate future FPGA debugging research.

In the rest of this paper, we first provide background on FPGAs and FPGA programming (§2). We then present the results of our bug study (§3), followed by the design of the collection of our static and dynamic analyses for debugging FPGA designs (§4). We provide implementation details of our analyses (§5), present evaluation results (§6), discuss related work (§7), and finally conclude (§8).

2 BACKGROUND

In this section, we discuss the FPGA development concepts that are necessary for understanding the bugs and debugging tools presented in this paper.

2.1 Languages for Hardware Programming

Developers program FPGAs by implementing a digital circuit in a hardware description language (HDL), such as Verilog [109], SystemVerilog [29], or VHDL [40]. HDLs enable developers to describe the behavior of a circuit in a cycle-by-cycle manner. For instance, the simple statement $c <= a + b$ subscribes to a broad programming paradigm: right hand expressions $(a + b)$ are computed and propagate to left hand operands (c) via an appropriate assignment operator (<=) at each clock cycle.

Emerging high level synthesis (HLS) tools enable hardware development using software programming languages, but impose significant performance and resource penalties compared to HDLs. For example, state-of-the-art HLS-implemented image processing is 6.6× slower and uses 5× more resources than an HDL-implementation [87]. As such, HDLs continue to dominate hardware development.

2.2 FPGA Debugging Stages

FPGA debugging contains two stages: simulation and on-FPGA testing. Simulation avoids lengthy hardware synthesis and is thus

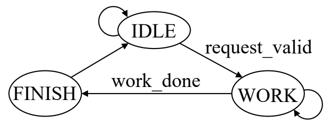

Figure 1: An example FSM with states represented by nodes, and transition conditions represented by edges. This FSM has three states: IDLE, WORK, and FINISH. A state transforms to another state when a certain condition is satisfied.

```verilog
reg [1:0] state;
always @(posedge clk) begin
  case(state)
    IDLE: if (request_valid) state <= WORK;
    WORK: if (work_done) state <= FINISH;
    FINISH: state <= IDLE;
  endcase
end
```

Listing 1: Verilog code implementing of the state transition of the FSM in Figure 1.

faster to iterate, but executes orders of magnitudes slower than on-FPGA testing [97]. In practice, developers simulate FPGA designs and iteratively fix any bugs they find before employing on-FPGA methods to test their design against more complex workloads (e.g., via stress testing).

2.3 FPGA Programming Techniques and Constructs

Hardware developers leverage a number of common techniques and constructs to implement FPGA designs.

Buffers and Queues. Hardware developers use buffers and queues to temporarily store values. Hardware buffers and queues are similar to their software equivalents, except they must be constant-sized, since all hardware components occupy a fixed area in a circuit.

Communication Control: Valid Interface. Logically, hardware circuits continually process data, with one or more input signals consumed every clock cycle. However, an input signal may not always be meaningful. For instance, a module may only receive a "packet" every 5 cycles. Thus, developers use valid interfaces that indicate whether a particular input is valid (i.e., a "valid bit" variable associated with one or more inputs).

Communication Control: Backpressure. In a communication channel where a source repeatedly sends data to a destination, the destination may use a backpressure or "ready" signal to inform the source that it needs time to process inputs. These signals indicate to the source that the destination can only receive x new packets, where x is defined by the communication protocol (e.g., $x = 1$ for a binary ready signal). In the event of backpressure, the source should stop sending packets or reduce the sending rate to avoid bugs at the destination.

Finite State Machines. Hardware developers frequently incorporate finite state machines (FSMs) in their designs [32, 118]. Figure 1 demonstrates an example FSM; Listing 1 shows the Verilog code that implements the FSM. In Verilog, an FSM is implemented using conditional assignments (e.g., an assignment inside a switch case); once a condition is satisfied, the "state" transfers along the arrows in the next clock cycle.

Module. A module is a sub-component of a Verilog circuit with a group of input and output signals, akin to a software function.

Intellectual Property (IP). A hardware intellectual property (IP) block is a "blackbox" module that implement commonly-used or platform-specific functionality, akin to a static software library. Like a software function, an IP block accepts user-controlled inputs and produces a set of outputs.

3 STUDY OF BUGS IN FPGA DESIGNS

To identify useful FPGA debugging tools, we study 68 hardware bugs across 19 FPGA designs and build a testbed [1] that reliably reproduces 20 of these bugs[1] in a push-button manner to enable their detailed study (§6.1). The study explores functional bugs, i.e., bugs in the HDL code that lead to functional issues rather than timing-related issues, since most production FPGA bugs are functional bugs [54]. Our methodology for gathering bugs is as follows:

Target Systems. First, we study bugs in four applications that we used in prior work. In particular, these applications use the Intel HARP platform [60], which uses the FPGA as a reconfigurable accelerator and provides an end-to-end acceleration stack. Specifically, we identify bugs in a SHA512 accelerator [24], Reed-Solomon decoder [25], and grayscale image accelerator [26] applications from HardCloud [45] (a framework with applications using HARP-based FPGA acceleration). Additionally, we find bugs in Optimus [85] (a HARP-based FPGA hypervisor).

Second, we examine bugs in hardware designs described in the ZipCPU website, a popular hardware design blog [2]. We identify bugs in SDSPI [3] (a library that drives an SD card through a Serial Peripheral Interface), Xilinx's two example AXI endpoint implementations [4, 5], and an FFT implementation [27].

Third, we study bugs found in hardware components from the most popular FPGA projects on GitHub, including a WiFi controller [6], a GPGPU processor [7], two RISC-V CPUs [8, 9], a Bitcoin Miner [10], a NIC [11, 12], and two hardware libraries [13, 14].

Finally, we examine a floating-point adder [15] that was provided to us by a hardware developer upon consultation about their experiences debugging hardware.

Bug Collection. Bugs in FPGA designs are difficult to collect, reproduce, and study due to the relative dearth of open-source hardware. Exacerbating this problem, among the 50 most popular FPGA projects on GitHub, 56% do not have a publicly-accessible bug tracker and 88% do not include test cases to reproduce bugs.

Therefore, rather than analyzing hardware bugs from bug trackers, we resorted to searching commit histories/issues of FPGA projects on GitHub to identify hardware bugs. In some cases, we found bugs through direct communication with developers (Optimus and FADD) and the ZipCPU website.

For each identified bug, we manually inspect related commit messages and discussions in GitHub Issues to understand the bug's root cause and symptoms. Sometimes, the commit messages and

[1]We select these 20 bugs because they occur in an application/platform with which we have familiarity. The rest of the bugs could be reproduced with additional effort.

Bug Class	**Bug Subclass**	**Number of Bugs**	**Common Symptoms**			
			App Stuck	**Data Loss**	**Incorrect Output**	**External**
Data Mis-Access	Buffer Overflow	5		✓		
	Bit Truncation	12			✓	✓
	Misindexing	5		✓	✓	
	Endianness Mismatch	1			✓	
	Failure-to-Update	5		✓	✓	✓
Communication	Deadlock	3	✓			
	Producer-Consumer Mismatch	3	✓	✓	✓	
	Signal Asynchrony	10			✓	
	Use-Without-Valid	1			✓	
Sementic	Protocol Violation	3	✓		✓	✓
	API Misuse	3			✓	
	Incomplete Implementation	7			✓	
	Erroneous Expression	10			✓	

Table 1: The result of our bug classification, including 3 main classes, 13 different subclasses, the number of bug instances observed in each subclass, and the common symptoms of each subclass.

issues do not provide sufficient information for a thorough understanding; in these cases, we inspect the hardware design's codebase as well as bug-related patches to understand the bug.

3.1 Bug Classification

We cluster bugs with similar root causes and symptoms into 3 main classes and 13 subclasses. Table 1 shows the classification results, identifying each bug subclass, the bug class to which each subclasses belongs, the number of bugs in the study that belong to each subclass, and the most common symptoms of each bug subclass.

The three bug classes roughly correspond to the three classes of software bugs from Li et al.'s software bug study [82] and are as follows: data mis-access bugs (§3.2), which arise when data is accessed without proper consideration for properties of the data format and are similar to software memory bugs; communication bugs (§3.3), which arise when a circuit violates inter-component communication standards and are similar to software concurrency bugs; and semantic bugs (§3.4), which arise from other remaining violations of a circuit's intended functionality and correspond to software semantic bugs. Some bugs could be classified into multiple classes/subclasses (e.g., a buffer overflow may arise because of an erroneous expression); we assign such multi-class bugs to the most related and specific subclass to which they could be assigned.

In the rest of this section, we provide a detailed description of each subclass of bug including their intrinsic properties, root causes, and common symptoms. We identify similarities between the hardware bugs and well-studied software bugs, which provide inspiration for the hardware debugging tools that we propose.

3.2 Data Mis-Access Bugs

Data mis-access bugs occur when the developer accesses data without proper considerations for size, endianness, and other properties of data. These bugs are similar to software memory bugs [82] (e.g., buffer overflows, our first example).

3.2.1 Buffer Overflow. A buffer overflow in an FPGA design occurs when a buffer is accessed with an offset that is greater than the size of the buffer. We identify 5 real-world examples of buffer overflow bugs in our bug study. We present a basic code snippet for simplicity.

```
1 reg mybuf [N-1:0]; // a buffer with N 1-bit elements
2 always @(posedge clk)
3   mybuf[offset] <= value; // offset >= N
```

Line 1 defines a buffer named `mybuf` consisting of N single-bit elements; `mybuf [N-1:0]` can be legally indexed from 0 to $N-1$ (inclusive). On Line 3, the snippet uses `offset` to assign a bit of `mybuf` to a `value`; however, the value of `offset` is greater than N and therefore overflows `mybuf`.

Accordingly, a buffer overflow in an FPGA design is similar to a software buffer overflow. However, unlike software buffer overflow bugs, which can corrupt memory by overwriting adjacent addresses, there is no notion of address adjacency beyond a buffer in hardware logic. Instead, hardware buffer overflows yield two possible outcomes: (1) the highest bits of `offset` are truncated, so an incorrect position in buffer is assigned (when the buffer size is a power of two), or (2) the assignment is ignored (when the buffer size is not a power of two). In select cases, hardware developers rely on truncation of the high bits of `offset` in their circuits for correctness, but this approach does not work for common data structures such as heaps and queues.
Symptoms. Data loss from truncation or ignored assignment.
Fixes. Hardware buffer overflows are fixed similarly to software buffer overflows: Developers enlarge the buffer or change the behavior of the FPGA design to avoid the overflow.

3.2.2 Bit Truncation. Bit truncation bugs in FPGA designs occur when assigning a variable to another variable with fewer bits. We identify 12 bit truncation bugs in 7 different FPGA designs.

The software equivalent of a bit truncation bug occurs when casting a variable to another variable that is represented with fewer bits. As in software, bit truncation in hardware may be used to intentionally discard part of a variable, which makes precise bug detection challenging.

In the following code snippet, `left` is a 42-bit variable and `right` is a 64-bit variable whose 42 bits from `[47:6]` contain meaningful data. On Line 4, `right` is cast into a 42-bit variable via `42'(right)` and then right-shifted by 6 bits before being assigned to `left`. As a result, bits [47 : 42] are truncated unintentionally.

```
1 reg [41:0] left;  // left is a 42-bit register
2 reg [63:0] right; // bits [47:6] are meaningful
3 always@(posedge clk)
4     left <= 42'(right) >> 6;
```

Symptoms. An incorrect value or an error (e.g., a page fault) reported by an external monitor (such as an FPGA shell).
Fixes. Depending on the developer's intentions, one technique for fixing truncation bugs is to perform shifts before bit-width casts. In our example, this means the developer would change Line 4 to: `left <= 42'(right >>6;)` Another potential fix is to grow the variables that can cause truncation. For instance, a developer can change the width of `left` to 48 bits, which prevents trucation of meaningful bits in `right`. In this case, Line 4 would be updated to: `left <= 48'(right) >> 6;`.

3.2.3 *Misindexing.*

A misindexing bug occurs when a developer uses an incorrect index to extract information from a variable. We identify 5 misindexing bugs in our study. For example, the IEEE-754 [22] standard defines the binary layout of 32-bit floating point, where the bits `[22:0]` are the fraction and the bits `[30:23]` are the exponent. However, in an implementation of floating point adder, the developer incorrectly extracted bits `[23:0]` as the fraction in a floating point adder, which lead to the wrong output value.
Symptoms. Incorrect output or data loss, if the misindexed data used for a control signal.
Fixes. Misindexing bugs are fixed by correcting the index.

3.2.4 *Endianness Mismatch.*

Endianness mismatches occur when an FPGA design assumes the wrong endianness for a particular piece of data (e.g., register arrays, off-chip DRAM, and disks), similar to how kernel code may assume the wrong endianness for device driver data. One instance of endianness mismatch bug is identified in our study.

In the simplified code snippet below, the circuit stores the least significant bits of an input in `data[7:0]` (on Line 2) and the most significant bits in `data[15:8]` (on Line 3). As a consequence, the input is stored in `data` in the little endian format. On Line 5, `data` is passed to a function expecting a big endian input, causing `out` to have the wrong result.

```
1 // Store data as little endian
2 data[7:0] <= least_significant_byte;
3 data[15:8] <= most_significant_byte;
4 // Pass data to function expecting big endian input
5 out <= big_endian_function(data);
```

Symptoms. A wrong value following assignment.
Fixes. Developers fix endianness mismatch bugs by manipulating bytes to account for the endianness difference. For example, the bug in the above code snippet is fixed by replacing Lines 2-3 with the following code:

```
1 data[7:0] <= most_significant_byte;
2 data[15:8] <= least_significant_byte;
```

3.2.5 *Failure-to-Update.*

A failure-to-update bug occurs when a developer forgets to put (including reset and initialization) a signal; we identify 5 failure-to-update bugs in our study.

Below, we provide a simple example code snippet of a failure-to-update bug. In this example, `input_counter` is incremented when the `input_valid` signal is set, while `output_counter` is incremented when `output_ready` is set. However, upon `reset`, only `input_counter` is set to 0, so `output_counter` may contain incorrect data after `reset`.

```
1 if (input_valid) input_counter <= input_counter + 1;
2 if (output_ready) output_counter <= output_counter + 1;
3 if (reset) input_counter <= 0;
```

Symptoms. Invalid output, data loss, or violation of communication interfaces if the failure-to-reset occurs on ready/valid signals (§2.3).
Fixes. The developer will reset each relevant signal in the system.

Takeaway #1. Data mis-access bugs can often be localized to a specific assignment, so stepping through dependency chains/FSM transitions can help localize the bug.
Takeaway #2. Data mis-access often results in data loss, so data loss detection (e.g., counting inputs received versus outputs sent) is crucial for finding bugs.

3.3 Communication Bugs

Communication bugs occur when the developer violates inter-component communication standards (e.g., inter-module interfaces, different clock domains, pipeline stages, etc.). They are similar to concurrency bugs in the software [82].

3.3.1 *Deadlock.*

A deadlock in an FPGA design occurs when two (or more) variables have a circular control dependency on each other. Hardware deadlocks are similar to software deadlocks, where a circular dependency among resources (e.g., locks) causes the program to stall. In hardware, deadlocks are triggered due to conditional assignments (e.g., assignments inside if-statements) that execute in parallel. We identify 3 deadlock bugs in our study.

In the following code snippet, if `a` and `b` are both initialized to 0, the assignment to `out` on Line 3 will never execute.

```
1 if (a) b <= 1;
2 if (b) a <= 1;
3 if (a) out <= result;
```

Symptoms. Infinite stall.
Fixes. To fix the bug in the above code snippet, a developer could initialize either `a` or `b` to 1. Fixing a deadlock bug in a complex circuit is often difficult because it is challenging to identify circular dependencies.

3.3.2 *Producer-Consumer Mismatch.*

When a collection of consumer registers cannot process the data values produced by a collection of producer registers, a producer-consumer bug occurs. For example, if the producers yield more valid data in a cycle than the consumers can process and store, data will be lost. Hence, a producer-consumer mismatch bug is similar to the classic "bounded-buffer" [51] producer-consumer problem in software, in which consumer threads can only process/store a limited quantity of output from producer threads. We identify 3 real-world examples of producer-consumer mismatch bugs in our study.

For a simple example, consider the following code snippet that uses a *valid interface* (§2.3), where producers generate and overwrite `x` and `y` at every cycle. If both `x_valid` and `y_valid` are *true* in the same cycle, then the value of `y` may be lost, since only the code on line 1 will execute.

```
if (x_valid) out <= x;
else if (y_valid) out <= y;
```

Symptoms. Data loss, invalid output, or an infinite stall (if the consumer FSM logic waits for a lost producer value).
Fixes. In software, locks and condition variables are used to force producer threads to wait until the consumer threads are ready to receive new values. In hardware, an analogous solution is to pause a producer by adding a back-pressure signal throughout the circuit. However, pausing a producer is invasive, since nearly all components of the circuit must be altered to accommodate the pause. Instead, an easier solution is creating a larger buffer for produced values that have not been consumed, assuming the maximum needed queue size is bounded.

3.3.3 *Signal Asynchrony.*

A signal asynchrony bug occurs when two variables that are supposed to be used together—such as a data variable and its valid/backpressure interface signals (§2.3) or the two operands of a mathematical operation—are not updated synchronously. We identify 10 signal asynchrony bugs in our study.

The following code snippet shows a simplified example of a signal asynchrony bug. The code responds to requests from a module that requires a minimum 2 cycle difference between requests and responses. Accordingly, upon receiving a request, the code buffers the response (calculated in a single cycle) in `buffered_response` for an extra cycle (Line 1), before outputting `final_response` (Line 2). Unfortunately, the `final_response_valid` signal (indicating the validity of the response data) is set immediately following receipt of `request` (Line 3), meaning `final_response` and `final_response_valid` are out of sync. For simplicity, we omit the code resetting `final_response_valid` to 0.

```
1  if (request) buffered_response <= input_data + 1;
2  final_response <= buffered_response;
3  if (request) final_response_valid <= 1;
```

Symptoms. An incorrect output value.
Fixes. The signal asynchrony bug in the snippet can be fixed by properly delaying the `final_response_valid` signal to be synchronous with the `final_response` signal. For instance, the developer may replace Line 3 with the following lines to fix the bug.

```
if (request) delayed_response_valid <= 1;
final_response_valid <= delayed_response_valid;
```

3.3.4 *Use-Without-Valid.*

A use-without-valid bug occurs when a data variable guarded by a valid signal (§2.3) is used when the valid signal is in an invalid state. Use-without-valid bugs are similar to signal asynchrony bugs, but occur when data is *used* erroneously, as opposed to signal asynchrony bugs which occur when data is *updated* erroneously. We identify one instance of use-without-valid bug in our study.

In the following code snippet, if `data` is a variable using a valid interface (e.g., with `data_valid` as its valid signal), `sum` may not be calculated correctly because it can use an invalid `data` as input.

```
1  // data is associated with a valid variable (data_valid)
2  sum <= sum + data;
```

Symptoms. An incorrect output value.
Fixes. Developers fix use-without-valid bugs by updating their code to use the correct valid interface. For example, the bug in the above code snippet is fixed by replacing Line 2 with the following two lines:

```
if (data_valid) sum <= sum + data;
else sum <= sum;
```

Takeaway #3. Given the proliferation of FSMs, circular dependencies, and infinite stalls in communication bugs, localizing the bugs would be easier with ability to record key states and statistics at arbitrary points in the circuit.
Takeaway #4. Like data mis-access bugs, debugging communications bugs would benefit from localized data loss detection.

3.4 Semantic Bugs

Semantic bugs occur due to remaining violations that cause the circuit to incorrectly perform its intended functionality. Semantic bugs include bugs where a developer does not correctly implement the entire high-level circuit specification (e.g., the protocol or FSM logic), misuses the API of a pre-implemented module, or does not implement special cases in complex logic. They are similar to semantic software bugs [82].

3.4.1 *Protocol Violation.*

Components of an FPGA design (e.g., modules) communicate through industry-standard communication protocols such as AXI4 [35]. However, such protocols are complex and contain corner cases that are difficult to cover in testing. If a developer fails to handle all cases correctly, a protocol violation occurs and escapes from simulation-based testing. We identify 3 instances of protocol violations.
Symptoms. Invalid outputs, infinite stall, or a protocol violation error reported by an external monitor (e.g., an FPGA shell).
Fixes. Fixing protocol violations requires correcting a mismatch between the high-level specification and implementation or adding logic for an unhandled corner case.

3.4.2 *API Misuse.*

FPGA designers use a hierarchy of modules to organize code and simplify the FPGA design process. An API misuse bug occurs when developers fail to use a pre-implemented module or IP block correctly. A hardware design may have an API misuse bug even if it implements all the involved communication protocols correctly, as it may pass wrong parameters to the module or configure it improperly. We identify 3 API misuse bugs in our study.

The following code snippet shows an example of an API misuse bug. Suppose that a developer wants to determine whether signal `a` is greater than signal `b` using a module, `greater_than`, which takes two parameters, `x` and `y`, and returns `x>y`. However, when instantiating the module, the developer erroneously connects signal `a` to the module's input port `y` and signal `b` to the module's input port `x`. Consequently, the module instance (i.e., `a_greater_than_b`) computes `b>a` instead of `a>b`, resulting in an incorrect output value.

```
// The greater_than module calculates whether x>y
greater_than a_greater_than_b(.x(b), .y(a), .result(out));
```

Symptoms. An incorrect output value.
Fixes. Fixing API misuse bugs involves correcting the mismatch between a module's API definition and how the module is used, usually by changing signal connections and the module's configuration.

3.4.3 *Incomplete Implementation.*

Hardware designs can be exceedingly complex, so hardware developers omit logic to handle corner cases, either intentionally or unintentionally. Such omissions are

incomplete implementation bugs and often occur in corner cases that are difficult to trigger during testing. We identify 7 instances of incomplete implementation bugs in our study.
Symptoms. Incorrect and invalid output.
Fixes. Developers fix incomplete implementation bugs by implementing the missing functionality, which may involve a redesign of certain components of the hardware design. Developers may also add additional test cases to cover the newly-added code.

3.4.4 Erroneous Expression. An erroneous expression bug occurs when hardware developers use a wrong expression in a control-flow statement (e.g., an if-statement) or data-flow statement (e.g., an assign-statement). Erroneous expression bugs are different from incomplete implementation bugs in that they involve an *incorrect* expression rather than *omitted* expressions. A wrong expression in a control-flow statement steers the hardware's control-flow to a wrong direction; a wrong expression in a data-flow statement generates an incorrect data value, which is used in other statements. In our study, we include 5 erroneous expression bugs in control-flow and 5 such bugs in data-flow.
Symptoms. Incorrect and invalid output.
Fixes. Developers fix erroneous expression bugs by correcting the erroneous expression in the control-flow or the data-flow.

Takeaway #5. Corner cases that trigger semantic bugs are difficult to detect, especially in simulation; runtime data recording enables debugging these scenarios.

4 DESIGN OF FPGA DEBUGGING TOOLS

Our bug study in §3 demonstrates that FPGA debugging can benefit from debugging tools similar to those used in software (e.g., flexible logging capabilities and program analysis). In contrast, past hardware debugging tools have emphasized airtight verification, and do little to help a developer diagnose the cause of a bug after its symptoms have been observed.

Therefore, we propose a set of hybrid static/dynamic analysis tools that simplify root cause diagnosis in FPGA designs. In this section, we describe the tools; the evaluation demonstrates their applicability to the bugs in our study (§6).

First, we unify simulation and on-FPGA debugging with SignalCat (§4.1). While "`printf`"-like statements have traditionally only been available in HDL simulators or required platform-specific IP to implement on FPGAs, SignalCat synthesizes these statements for actual FPGA deployments across multiple platforms. The infrastructure provided by SignalCat serves as a cornerstone upon which developers can build symptom-specific tools without needing to consider the execution context of the circuit and applies directly to all 5 of the takeaways from our bug study.

Using SignalCat, we build three monitoring tools that gather targeted information based upon insights from our bug study. First, FSM Monitor (§4.2) statically detects FSM variables and records them at runtime, automatically reconstructing FSM state-transition traces to aid developers in debugging. Second, Dependency Monitor (§4.3) statically analyzes the dependencies of user-specified variables and dynamically records the updates to each dependency, allowing developers to backtrace and localize the source of an incorrect output-of-interest. Third, Statistics Monitor (§4.4) provides counters for user-specified events, helping users identify bugs reflected in statistical metadata (e.g., data loss is often indicated by fewer outputs generated than inputs received).

Finally, given the commonality of data loss in our bug symptoms, we develop an additional tool for the event that a developer suspects or detects data loss. In particular, LossCheck (§4.5) pinpoints the location of data loss within a hardware design. LossCheck statically analyzes an FPGA design and instruments it with logic that dynamically checks for data loss in suspected locations.

4.1 SignalCat for Unified Logging

Our bug study shows that hardware debugging would benefit from the ability to log arbitrary runtime information, just as software debugging does [121]. Today, while developers can use debug statements (e.g., `$display`) to log values during HDL simulation, similar tools are not pervasively available on deployed FPGAs without specific FPGA virtualization or IO support [80, 97]. In lieu of generic "`printf`"-like statements, developers typically use vendor-provided data recording IPs (e.g., Intel SignalTap [62] and Xilinx ILA [119]) to record a subset of variables when debugging a deployed FPGA design. Thus, developers must maintain two different versions of their FPGA design when debugging, one that uses simulation-based deubgging primitives, and one that uses on-FPGA primatives.

SignalCat bridges this gap by unifying simulation-based and on-FPGA debugging through automatic generation of on-FPGA recording logic (e.g., using FPGA vendors' IPs) from debugging statements (e.g., `$display`). SignalCat incorporates a static and a dynamic component. The static component analyzes the path constraints of debugging statements and generates an IP instance for on-FPGA data collection, while the dynamic component records the trace via the IP instance in an on-FPGA scenario.

SignalCat searches the abstract syntax tree (AST) of an FPGA design for debugging statements. For each such statement, SignalCat determines the arguments (i.e., the variables that the developers want to print) and the path constraint (i.e., the conditions under which the statement is reached) of the statement. Then, SignalCat generates an instance of a vendor-provided data recording IP to record the collected arguments and path constraints, encoding path constraints as a 1-bit `bool` per debugging statement. At each cycle, The system stores all arguments and encoded path constraints in the recording IP buffer if at least one path constraint is true. SignalCat reconstructs and prints debugging logs after execution allowing the same format for on-FPGA debugging and simulation.

SignalCat requires that developers specify the size (i.e., the number of data entries) of the IP's recording buffer and events that start and stop data recording (e.g., when the first packet arrives or an assertion is triggered). Developers can also configure the buffer to capture a fixed interval before and/or after the user-provided event.

Since SignalCat provides a single interface for simulation and on-FPGA logging, developers of debugging tools can instrument an HDL design with a "`printf`"-like statement and support simulation and on-FPGA debugging with a single code-base. In fact, all of our subsequent debugging tools (§4.2–§4.5) leverage SignalCat for runtime data recording.

4.2 FSM Monitor for State Machine Traces

Hardware circuits often use finite state machines (FSMs) in their design (§2.3). When this design paradigm is used, an FSM (state-transition) trace provides a user-friendly abstraction for circuit execution and debugging, especially in comparison to a low-level waveform (i.e., a graph of all signals at every cycle). Therefore, we propose FSM Monitor to help developers automatically generate FSM traces. FSM Monitor detects FSMs in a circuit and generates logic that monitors state changes for each detected FSM.

Hardware FSMs employ fixed code patterns that are detectable with static analysis [32, 118], unlike software FSMs, which are difficult to detect without complex online tracing tools [39]. In an FSM, a state transforms to another state when certain condition(s) are satisfied. State transitions usually conditionally assign (e.g., an assignment inside a switch case) to FSM variables and include FSM variables as a part of the condition. Additionally, circuits rarely perform mathematical operations (e.g., addition or subtraction) on FSM variables and rarely select individual bits of FSM variables.

Accordingly, FSM Monitor traverses the abstract syntax tree (AST) of a circuit and searches for FSM variables by using the aforementioned heuristics. For each identified FSM variable, FSM Monitor generates Verilog code that displays a log message when the variable is updated.

FSM Monitor's heuristics can incur both false positives and false negatives, but we find a high degree of accuracy in our evaluation (of the 32 manually-identified FSMs in our benchmark suite, FSM Monitor has 0 false positives and 5 false negatives). Furthermore, more sophisticated FSM detection approaches, like those used by the Intel and Xilinx synthesizers, could further increase accuracy. Finally, FSM Monitor allows developers to patch mistakes by adding undetected FSMs and filtering out FSMs that are inaccurate or irrelevant for their current bug.

4.3 Dependency Monitor for Provenance Tracking

Our bug study indicates that the only symptom of many hardware bugs is one or more incorrect output values (Table 2). Since the root cause of a bug can occur many cycles prior to output generation, it is useful to build the dependency chains for a specific variable and trace updates to variables in the dependency chain during execution.

We therefore build Dependency Monitor to statically analyze the dependencies of a variable and generate the necessary logic to monitor their updates. Dependency Monitor first statically finds all registers that may propagate to a variable v within the previous k cycles (where v and k are specified by the developer). Dependency Monitor then generates logic that logs each update to variables in the dependency chain at runtime.

Dependency Monitor handles partial assignments (i.e., assignment to a strict subset of a variable's bits) by logically splitting a partially assigned variable to multiple variables. Similarly, Dependency Monitor splits constant-indexed arrays into individual variables. If an array is accessed with at least one variable index, Dependency Monitor considers the whole array as an individual register and an assignment to/from the array as a special assignment that only occurs when the index matches. To track dependencies through a blackbox IP, Dependency Monitor requires the developer to provide a model of data and control dependencies within the IP. An IP model describes the relationship between the input signals and the output signals of the IP, which is included in the IP specification and typically well-understood by developers before using an IP. Developers can reuse IP models across projects that share the same IPs.

By default, Dependency Monitor analyzes both control and data dependencies; however, it can be configured to only analyze one type of dependency.

4.4 Statistics Monitor for Counting Events-of-Interest

Collecting hardware statistics (i.e., event counters) provides insight into program execution without requiring cycle-by-cycle recording of numerous variables. Furthermore, per-component (e.g., per pipeline stage) counters help a developer localize a statistical anomaly (indicative of a bug) to a small region of a complex circuit.

Accordingly, we propose Statistics Monitor, a tool to help developers collect statistics for events of interest when debugging an FPGA design. Statistics Monitor generates Verilog code that counts occurrences of single-bit signals specified by a developer and adds logging code that emits messages when counts change.

Statistics Monitor is particularly useful when developers suspect that 1) it is too expensive (i.e., with regard to resource consumption) or unnecessary to record all variables of interest on an FPGA deployment (especially cycle-by-cycle), and 2) the bug's symptoms can be inferred via statistical anomalies (e.g., unexpected differences between valid input and valid output counts, indicating potential data loss).

4.5 LossCheck for Precise Data Loss Localization

While Statistics Monitor may indicate the presence of data loss (among other bug symptoms) and may localize it to a portion of the circuit, the pervasiveness of bugs manifesting as data loss in our bug study indicates that precise data loss localization would be helpful for hardware debugging.

We therefore design LossCheck, a tool that localizes the root cause of data loss symptoms. A developer specifies a SOURCE register, a SINK register, and a valid signal for SOURCE (§2.3). Then, LossCheck instruments the HDL code to monitor the propagation of valid data between SOURCE and SINK. If a valid register is overwritten before its value is propagated from SOURCE to SINK (i.e., overwritten before being used as a right-hand variable), LossCheck indicates potential data loss.

We note that the tracking of data propagation logic in LossCheck shares similarities with that of Dependency Monitor. However, unlike Dependency Monitor, LossCheck does not yield a trace of updates to variables of interest in a dependency chain. Rather, LossCheck indicates the precise location of a potential data loss. Ultimately, LossCheck's dynamic analysis conveniently enables automatic localization of data loss bugs without recording a large number of data propagation events.

We now describe how LossCheck statically analyzes HDL code (§4.5.1), instruments the code (§4.5.2), and dynamically detects data loss while mitigating false alerts (§4.5.3). We then discuss the limitations of LossCheck (§4.5.4).

4.5.1 Static Analysis of Data Propagation. LossCheck statically analyzes data propagation in an FPGA design and builds a table of *propagation relations*. It uses these relations to calculate metadata variables that indicate potential data loss (§4.5.2).

A propagation relation $X \rightsquigarrow_\sigma Y$ implies that the data value stored in register X will propagate to register Y when the condition σ is satisfied. In other words, the value stored in Y at cycle $k+1$ (i.e., Y_{k+1}) will be influenced by the value stored in X at cycle k (i.e., X_k), if σ is true at cycle k (i.e, σ_k).

At a high level, LossCheck uses logic similar to Dependency Monitor to detect propagation relations and thereby build the propagation relation table. More specifically, LossCheck first identifies a set of data propagation sequences through which a value stored in Source can propagate to Sink. LossCheck then analyzes the control and data dependencies for each register R in the propagation sequences, and adds each identified propagation relation into the table.

We use the following code snippet as a running example of how LossCheck works, where `in` is the Source register, `out` is the Sink register, and `in_valid` is the valid bit for `in`:

```
1  always @(posedge clk) begin
2    // buggy code (b's value can be lost)
3    if (cond_a) out <= a;
4    else if (cond_b) out <= b;
5    if (in_valid) b <= in;
6  end
```

To analyze the dependencies of b in this example, LossCheck first detects the propagation sequence: `in` $\longrightarrow$ `b` $\longrightarrow$ `out`. LossCheck then analyzes the dependencies for `b` and `out`, building the following table with 3 propagation relations.

Line	Propagation Relations
3	$\texttt{a} \rightsquigarrow_{\texttt{cond\_a}} \texttt{out}$
4	$\texttt{b} \rightsquigarrow_{\neg\texttt{cond\_a}\wedge\texttt{cond\_b}} \texttt{out}$
5	$\texttt{in} \rightsquigarrow_{\texttt{in\_valid}} \texttt{b}$

Similar to Dependency Monitor, if the source code for an IP is unavailable, LossCheck inserts propagation relations into the table based upon developer-provided IP models.

4.5.2 Instrumentation of HDL Code. LossCheck uses the propagation relations to guide circuit instrumentation that enables data loss detection at runtime. The instrumentation process of LossCheck contains two phases: 1) inferring various loss-related metadata for each register in each propagation sequence, and 2) inserting corresponding logic to check for potential data loss via this metadata.

Assignment, Validity, and Propagation Statuses. Intuitively, potential data loss occurs when the *assignment* of a *valid* register occurs before its value is *propagated* to another register, thereby overwriting (unused) valid data. So, to detect potential data loss for a register R, LossCheck generates assignment $A(R)$, valid-assignment $V(R)$, and propagation $P(R)$ shadow variables for the register.

For some cycle k, a register's assignment status $A(R)_k$ indicates whether R is assigned a value during cycle k. The value of $A(R)_k$ is inferred at runtime from the propagation relation table. Specifically, $A(R)_k$ evaluates to *true* if at least one register R' propagates its value to R at cycle k. More formally, the condition σ for some propagation relation $R' \rightsquigarrow_\sigma R$ must be satisfied at cycle k.

Similarly, $V(R)_k$ indicates whether R is specifically assigned a *valid* value during cycle k. $V(R)_k$ is therefore determined by combining the logic for calculating $A(R)_k$ with runtime information about data validity. In simple cases (such as our code example), data validity status is trivially available for the variable of interest (e.g., via a corresponding valid signal); in more complex cases, LossCheck calculates validity status for each variable of interest according to the initial input validity value and propagation relations.

Finally, a register's propagation status $P(R)_k$ indicates whether R is used to compute another register's value during cycle k. Similar to how $A(R)_k$ represents assignment *to* register R, $P(R)_k$ represents assignment *from* register R. Thus, $P(R)_k$ evaluates to *true* if R can propagate its value to at least one register R' at cycle k (i.e., if the condition σ for some propagation relation $R \rightsquigarrow_\sigma R'$ is satisfied at cycle k).

After LossCheck determines the values of $A(R)$, $V(R)$, and $P(R)$, it instruments the circuit with the logic to compute the values of these variables at each cycle. Below, we apply these rules to variable `b` from the original code snippet:

```
1  always @(posedge clk) begin
2    // update shadow vars for next cycle
3    A_b <= in_valid;
4    V_b <= in_valid;
5    P_b <= ~cond_a & cond_b;
6  end
```

Lines 3–5 calculate the values of $A(\texttt{b})$, $V(\texttt{b})$, and $P(\texttt{b})$ for the next cycle based on the propagation relations. We note that, in this example, $A(\texttt{b}) = V(\texttt{b})$ because assignment to `b` is guarded by the valid signal `in_valid`.

Inserting Checking Logic. Given a register's shadow variables, data loss for register R at cycle k occurs if the following 3 conditions hold: (1) R is assigned at cycle k—i.e., $A(R)_k = \textit{true}$, (2) R is not simultaneously propagated at cycle k—i.e., $P(R)_k = \textit{false}$, and (3) R was assigned a valid value in some previous cycle, which has not yet propagated.

The first two conditions are trivially calculated for R at the current cycle via aforementioned logic. For the third condition, LossCheck keeps track of an additional "Needs-Propagation" variable $N(R)$, which is set to *true* when a valid value is assigned to R and reset to *false* when the value propagates. In mathematical terms, $N(R)_0 = \textit{false}$ (since no valid value has been assigned at cycle 0), and for $k > 0$,

$$N(R)_k = V(R)_{k-1} \vee [N(R)_{k-1} \wedge \neg P(R)_{k-1}] \ . \tag{1}$$

Potential data loss at cycle k is then calculated as:

$$\texttt{Loss} = A(R)_k \wedge \neg P(R)_k \wedge N(R)_k \ . \tag{2}$$

Notably, while the shadow variables (i.e., $A(R)$, $P(R)$, and $N(R)$) have a unique value at each cycle, k, LossCheck can detect loss in R at cycle k using only the most recent value of each shadow variable, (i.e.,$A(R)_k$, $P(R)_k$, and $N(R)_k$). Consequently, the amount of state that LossCheck tracks is bounded, so LossCheck can be realized on hardware.

LossCheck generates code that calculates $N(R)$ and checks Equation 2. The instrumented circuit that checks for data loss on `b` is:

```
always @(posedge clk) begin
  // calculate N_b for next cycle from shadow vars
  if (reset) N_b <= 0;
  else N_b <= V_b | (N_b & ~P_b);
  // check for data loss at current cycle
```

```
6     if (A_b & ~P_b & N_b)
7       $display("LossCheck: potential data loss at b");
8   end
```

Lines 3–4 calculate $N(\mathrm{b})$ for the next cycle according to Equation 1, and Lines 6–7 perform the check for potential data loss at cycle k based on Equation 2.

4.5.3 Filtering False Positives and Final Analysis. Notably, LossCheck's design can generate false positives due to an intentional data *drop* (as opposed to an unintentional data *loss*). For example, an FPGA may intentionally drop a network packet input that fails a checksum; LossCheck would flag the packet as data loss. Accordingly, LossCheck uses an FPGA design's test cases—presumably passed during simulation testing— as "ground-truth" test programs; LossCheck suppresses warnings triggered by these test cases. We note that pre-existing test programs for the open-source designs in our study filter 23/24 false positive registers (i.e., those with intentional data drops).

Like the monitors, LossCheck leverages SignalCat to transform the filtered debugging statements (indicating unintentional data loss) into log messages for either simulation or on-FPGA scenarios. Thus, if potential data loss is detected for some register R, a log message indicates R as the source of the loss, and the bug can be precisely localized.

4.5.4 Limitations of LossCheck. While LossCheck can accurately localize data losses to a specific register, it cannot distinguish intentional data drops from unintentional data losses. As a consequence, if an unintentional data loss and an intentional data drop occur at the same place, the data loss may be filtered by LossCheck, resulting in a false negative. We identify a single such false negative (out of 7 data loss bugs) in our testbed (§6.3).

5 IMPLEMENTATION

We build our static analyses using Pyverilog [108], a toolbox for Verilog analysis and instrumentation. We use Pyverilog's dataflow analysis framework to analyze data dependencies and its Verilog code generator to output the instrumented circuit. Furthermore, to analyze circuits developed in SystemVerilog (i.e., an extension of Verilog with more language features), we augment Pyverilog to use the more modern SystemVerilog parser of Verilator [105], a SystemVerilog simulator. Verilator parses SystemVerilog files and performs optimizations such as inline expansion and module instantiation, resulting in an analysis-friendly abstract syntax tree (AST) that Pyverilog can analyze. We modify and add 269 lines of C++ code and 1,750 lines of Python code to integrate Verilator and Pyverilog.

We implement the debugging tools (i.e., SignalCat, FSM Monitor, Dependency Monitor, Statistics Monitor, and LossCheck) as a collection of analysis and instrumentation passes on Pyverilog ASTs. These passes are implemented with 3,797 lines of Python code.

Dependency Monitor and LossCheck require developers to implement a model that describes the relation between the inputs and outputs for each closed-source IP. In our testbed, three IPs are used: `altsyncram`, a block RAM implementation; `scfifo`, a single clock queue implementation; and `dcfifo`, a double clock queue implementation. We implement the models for these IPs in Python and Verilog, resulting in 394 lines of code in total.

6 EVALUATION

In this section, we first present our testbed (§6.1) and experimental setup (§6.2). Then, we evaluate the effectiveness of our debugging tools at helping developers debug the bugs in our study (§6.3). Finally, we present the resource usage and performance overhead when using the debugging tools to diagnose the study bugs (§6.4).

6.1 Testbed of Reproducible FPGA Bugs

We built and released a testbed consisting of 20 bugs that we reproduced to facilitate further study of FPGA bugs and FPGA debugging tools [1]. The bugs span the 3 major classes of bugs we identified—data mis-access, communication, and semantic—and multiple development platforms (e.g., Intel HARP and Xilinx). For each bug, we identify the subclass, application, symptom, and the tools that are helpful when debugging each bug, as shown in Table 2. The artifact also includes a simplified code snippet for each bug for explanation purposes and provides instructions for reproducing the bug in a push-button manner with the open-source Verilator simulator [105] . Using a simulator eliminates the need for testbed users to spend substantial time and effort acquiring design-specific knowledge that would otherwise be necessary to reproduce each bug.

Although each bug in the testbed is reproducible on real hardware, but, we opt to reproduce the bugs in Verilator for 3 reasons. First, a Verilator-compatible testbed demonstrates that both the fundamental properties of the bugs and the logic of our debugging tools are broadly-applicable in FPGA development. Second, other developers can reason about these bugs and a range of development platforms without purchasing expensive hardware. Third, Verilator simplifies the environmental conditions required to reproduce each bug—crucially, without changing the buggy programs themselves. For the key platform-specific recording IP primitives used by SignalCat (SignalTap [62] and ILA [119]), we provide support for simulating their behavior. Unless specifically mentioned, the buffer size for these data recording IPs is fixed at 8,192 entries.

6.2 Experimental Setup

Platform for Overhead Measurement. We evaluate the resource and performance overhead of our debugging tools using Quartus 17.0 [17] and Vivado 2020.2 [19], the official synthesizers for Intel's and Xilinx's FPGAs, respectively. We synthesize all Intel HARP-specific designs to the HARP platform [60] (using Quartus), with the remaining designs synthesized to the Xilinx KC705 [21] platform (using Vivado).
Use Cases. We evaluate our tools in two use cases. In the first case, we use SignalCat and the three monitors (FSM Monitor, Dependency Monitor, and Statistics Monitor) to debug all bugs in our study; in the second one, we use LossCheck to localize the source of data loss symptoms for the 4 relevant bugs. Table 2 shows the tools used during the debugging process of each bug.

6.3 Effectiveness of Debugging Tools

We evaluate the effectiveness of our debugging tools by assessing how much they simplify root cause diagnosis for the bugs in our study (§3). An experienced developer could diagnose, localize, and

ID	Subclass	Application	Platform	Symptom				Helpful Tools				
				Stuck	Loss	Incor.	Ext.	SC	FSM	Stat.	Dep.	LC
D1	Buffer Overflow	RSD	HARP	✓	✓			✓	✓	✓		✓
D2		Grayscale	HARP	✓	✓			✓	✓	✓		✓
D3		Optimus	HARP	✓	✓			✓	✓	✓	✓	✓
D4		Frame FIFO	Generic		✓	✓		✓	✓			✓
D5	Bit Truncation	SHA512	HARP	✓			✓	✓	✓		✓	
D6		FFT	Generic			✓		✓			✓	
D7	Misindexing	FADD	Generic			✓		✓				
D8		AXI-Stream Switch	Generic			✓		✓				
D9	Endianness Mismatch	SDSPI	Generic			✓		✓				
D10	Failure-to-Update	SHA512	HARP				✓	✓	✓		✓	
D11		Frame FIFO	Generic		✓			✓			✓	
D12		Frame FIFO	Generic			✓		✓			✓	
D13		Frame Length Measurer	Generic			✓		✓		✓	✓	
C1	Deadlock	SDSPI	Generic	✓			✓	✓			✓	
C2	Producer-Consumer Mismatch	Optimus	HARP	✓	✓			✓	✓	✓	✓	✓
C3	Signal Asynchrony	SDSPI	Generic			✓		✓				
C4		AXI-Stream FIFO	Generic		✓			✓				✓
S1	Protocol Violation	AXI-Lite Demo	Xilinx				✓	✓				
S2		AXI-Stream Demo	Xilinx				✓	✓				
S3	Incomplete Implementation	AXI-Stream Adapter	Generic			✓		✓				

Table 2: The testbed of reproducible bugs, including their classes, subclasses, platforms, symptoms, and which of our new tools help localize their root cause. Bug D1–D13 are data mis-access bugs, Bug C1–C4 are communication bugs, and Bug S1–S3 are semantic bugs. A "Generic" platform means that the application does not target on a specific platform and can be synthesized to different FPGAs. For bug symptoms, "Stuck" indicates a symptom of infinite waiting; "Loss' indicates a data loss; "Incor." means the FPGA design gives an incorrect output; and "Ext." means an external monitor (such as an FPGA shell) reports an error. For helpful tools, "SC" stands for SignalCat; "FSM" stands for FSM monitor; "Stat." stands for statistics monitor; "Dep." stands for dependency monitor; and "LC" stands for LossCheck.

fix the bugs in our study without extra tooling; this is what occurred when these bugs were first reported. But, we find that the localization process is simpler when using our tools. We specifically answer two questions (1) How often is each tool useful when debugging the bugs in our study? and (2) How much work do the debugging tools automate? Additionally, we provide a case study that demonstrates how a developer would use the tools to localize a data loss bug in an Intel HARP application.

SignalCat and Monitors. SignalCat is useful for debugging every bug in our study, serving as the fundamental cross-platform logging infrastructure. Each of the 3 monitors assists with debugging at least four bugs from the testbed. During debugging (with SignalCat and the 3 monitors), we often find FSM Monitor to be the most helpful in an initial debugging iteration when one or more FSMs were present in the design. Statistics Monitor is generally most usefully deployed in subsequent iterations, where developers try to narrow down the search space of a bug's root cause. Finally, upon encountering a variable with an unexpected value, SignalCat is useful for directly recording updates to the specific variable, while Dependency Monitor supplements this with an analysis of the variable's dependencies. On average, SignalCat and the monitors generate and insert 72 lines of Verilog code to help with root cause localization.

LossCheck. LossCheck precisely locates the root cause of data loss (i.e., a specific register) for 6 out of 7 bugs exhibiting data loss (i.e., Bugs D1, D2, D3, D4, C2, and C4) in our study. For 2 of these bugs (D4 and C4), LossCheck uniquely identifies the root cause of the bug without using the false positive filtering technique in §4.5.3. For 3 of these bugs (D2, D3, and C2), LossCheck uses the false positive filtering technique to localize the bug without reporting false positives. For the Reed-Solomon decoder buffer overflow (D1), LossCheck reports 1 false positive (i.e., it mistakenly identifies an intentionally dropped register as unintentional data loss), because the developer-provided test case does not perform an intentional data drop at the mis-reported register, so LossCheck does not silence the warning. LossCheck cannot localize the data loss in Bug D11 because the unintentional data loss occurs in a register where the data value may be dropped intentionally under certain conditions; as a result, the data loss is mis-filtered by the LossCheck's false positive filtering. LossCheck generates and inserts 522–19,462 lines of Verilog code to analyze data propagation and detect data loss at runtime, which helps developers avoid the time-consuming manual implementation of data loss checking logic.

Case Study: Debugging Grayscale's Buffer Overflow. We describe a case study in which a developer uses the new tools to debug a buffer overflow in the Grayscale application [26]. Grayscale is an end-to-end application written for Intel HARP [60] that includes an FPGA accelerator and a software component. The CPU-side software component reads an image from the file system and programs the FPGA accelerator to read the image from CPU-side memory, perform the grayscale transformation, and write the result back to CPU-side memory. The software component identifies that the acceleration task hangs when the bug occurs.

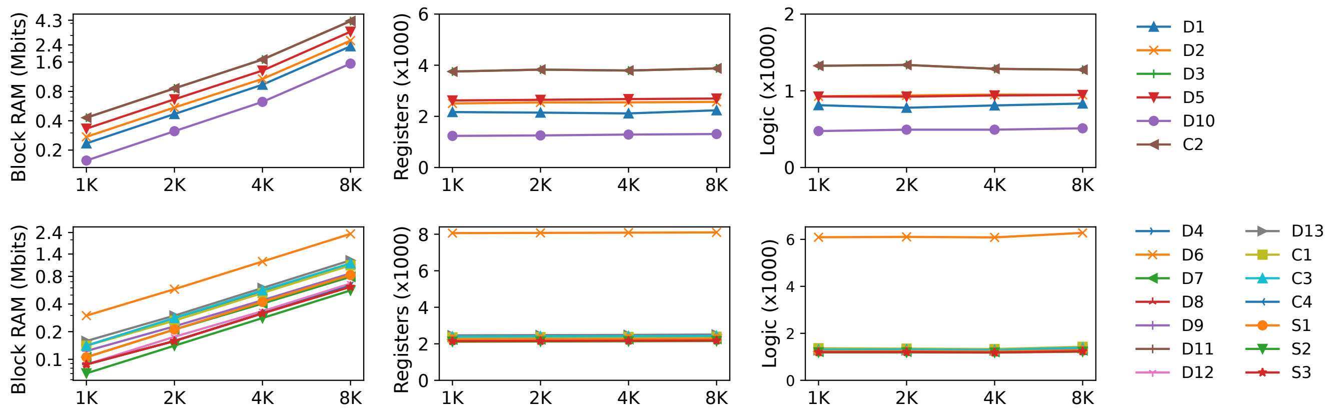

Figure 2: The resource overhead of manual debugging using SignalCat, FSM Monitor, Statistics Monitor, and Dependency Monitor on Intel HARP (top) and Xilinx KC705 (bottom) platforms. Resource overheads (y-axes) are shown in terms of block RAM, registers, and logic (i.e., the three types of resources on an FPGA) with an increasing recording buffer size (x-axes). The buffer size and block RAM overhead are shown in log-scale.

Grayscale consists of multiple FSMs, so the developer first uses FSM Monitor to identify the state of each FSM when the hang occurs. The developer re-executes the application to trigger the bug. FSM Monitor's output identifies that the accelerator finished reading data from the CPU, since the `read` FSM—which controls how the accelerator reads CPU memory—is in the `RD_FINISH` state. However, the circuit has not finished writing data to the CPU, since the `write` FSM—which controls how the accelerator writes CPU memory—is in the `WR_DATA` state. The developer concludes that the hang occurs in write-related logic.

Next, the developer inspects the state transition logic of the `write` FSM. They find that the state of the write FSM only transfers from `WR_DATA` to `WR_FINISH` after the accelerator writes the whole transformed image to the CPU-side memory. Since the accelerator has already read all data from the CPU (i.e., the `read` FSM is in the `RD_FINISH` state), the hang indicates data loss in the accelerator during the propagation between a memory read and its corresponding memory write.

Finally, the developer uses LossCheck to identify the source of the data loss. They re-execute the application with LossCheck enabled. LossCheck identifies the source of the data loss as a specific register in the accelerator.

6.4 Efficiency of Debugging Tools

In this section, we assess the efficiency of the debugging tools by measuring (1) the additional resources consumed when circuits are instrumented using our tools—i.e., the resource overhead, and (2) the necessary clock frequency slowdown stemming from the augmented logic that must execute each cycle—i.e., the runtime performance overhead.

SignalCat and Monitors. Figure 2 shows the resource overhead (in terms of block RAM, registers, and logic) of SignalCat and the monitors, applied to each buggy design. The most significant resource overhead lies in block RAM usage, which increases linearly as the developer-specified recording buffer size increases. The register and logic overheads tend to be stable for each bug, regardless of the recording buffer size. Among our benchmarks, the two bugs on the Optimus hypervisor and the Bit Truncation bug on the FFT accelerator incur the largest register and logic overheads consuming approximately 0.23% and 0.3% of register and logic resources on the Intel platform (3.08% and 1.99% on Xilinx).

Runtime performance overhead is only incurred for 1 design; namely, Optimus fails to achieve its targeted clock frequency (400 MHz) after the debugging instrumentation. As a result, we reduce its frequency to 200 MHz for debugging. While SHA512 also targets a 400 MHz frequency, it still achieves this frequency after instrumentation. Other designs target a 200 MHz frequency and likewise do not incur performance overhead to account for debugging logic.

LossCheck. Figure 3 shows LossCheck's resource overhead in terms of registers and logic for the data loss bugs in our study. LossCheck's instrumentation uses less than 1.7% of the total register and logic resources for the four data loss bugs on the Intel platform, and uses less than 0.7% of total resources for the two data loss bugs on Xilinx.

As with SignalCat and the monitors, LossCheck reduces the frequency of Optimus from 400 MHz to 200 MHz. The 200 MHz target frequency of other FPGA designs remain unchanged.

7 RELATED WORK

Hardware Bug Studies. HardFails [50] performs a bug study of security bugs in CPUs that include real-world and synthetic bugs and creates a testbed by injecting bugs into an open-source CPU design. HardFails only includes security bugs, which are representative of few bugs that make it to production [54]. In contrast, our study examines real-world functionality bugs in FPGA designs.

Simulation-Based FPGA Debugging. Developers usually simulate an FPGA design before deploying on-FPGA. Most simulators [18, 20, 105, 107, 114] can generate a waveform—a visualization of signal values—during simulation to aid with debugging. Previous

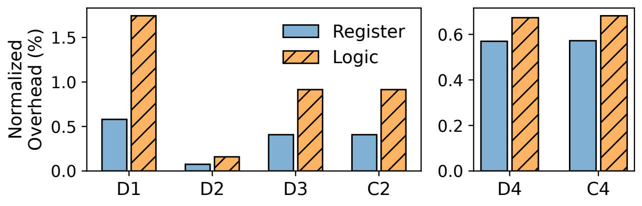

Figure 3: LossCheck's overhead in terms of registers and logic, normalized to the total resources available on Intel HARP (left) and Xilinx KC705 (right) platforms.

research accelerates simulation-based debugging using language features [48, 89] and by offloading simulation to an FPGA [64, 65, 97] or a GPU [91]. Our debugging tools are designed for both on-FPGA and simulation-based debugging.

Trace-Based FPGA Debugging. Trace-based FPGA debugging tools allow developers to collect the value of a selected set of signals in an FPGA deployment. FPGA vendors provide IPs (e.g., Intel SignalTap [62] and Xilinx ILA [119]) that export manual interfaces (e.g., GUIs). To use these tools, developers manually specify the signals that they wish to trace and triggering conditions that should enable tracing output. In contrast, SignalCat automates the selection of signals and corresponding trigger conditions (by statically analyzing "`printf`"-like statements and their path constraints) and provides a natural, vendor-agnostic debugging interface. Prior work reduces the runtime recording overhead of platform-specific IPs by reducing buffer usage [55–57, 59, 76, 84, 90, 103]; SignalCat can benefit from these optimizations when applicable.

Checkpointing-Based FPGA Debugging. Checkpointing-based FPGA tools [37, 38, 75, 80, 97] allow a developer to capture the state of an FPGA deployment for later analysis or debugging, but do not help with localizing the root cause of bugs. Our debugging infrastructure could benefit from similar checkpoint-based functionality.

Synthesizing Traditionally-Unsynthesizable HDL. Cascade [97] and Synergy [80] enable traditionally "unsynthesizable" Verilog, including "`printf`"-like statements, to execute on an FPGA. Cascade and Synergy can store arbitrarily-long logs in off-FPGA storage (e.g., in CPU-side memory or disk), but may slow down the circuit since they pause circuit execution when executing "`printf`"-like statements. In contrast, SignalCat offers a different tradeoff: SignalCat imposes lower overhead since it does not pause circuit execution, but can only store limited information since it uses on-FPGA storage (e.g., block RAM).

Interactive FPGA Debugging. Interactive FPGA debugging tools allow a developer to interactively manipulate packets in their FPGA's communication channels [86] and provide GDB-like interfaces for FPGA debugging [34]. These tools are useful during simulation but are not applicable for on-FPGA debugging and do not directly help a developer localize the root-cause of a hardware bug.

Traditional Hardware Testing. Traditionally, hardware developers implement test suites with industry standard frameworks [30] to extensively test hardware designs in simulation. Hardware fuzzing techniques [79, 110] and formal verification [58, 63, 81, 93, 94, 115, 126] help developers find and eliminate bugs before fabrication, but do not help a developer identify the root-cause of a bug and are resource-intensive. In contrast, our work explores bug localization tools designed for both simulation and on-FPGA scenarios.

Hardware-Assisted Testing and Debugging. A plethora of tools [49, 66–69, 130] have used efficient hardware tracing techniques (typically used in profiling and optimization of hardware/software designs [70–72]) for testing and debugging. In this paper, we show how reconfigurable hardware can be leveraged to instead design more targeted debugging support by designing and implementing foundational debugging tools. We expect future work to use the reconfigurable nature of FPGAs to design advanced debugging support.

Software Bug Detection at Runtime. Our work on FPGA bug localization is inspired by software debugging tools and techniques such as AddressSanitizer [98], ThreadSanitizer [99], Memcheck [100], and dynamic slicing [95]. Particularly, LossCheck's key building block–tracking data propagation dynamically—is closely inspired by such work. Since our own work shows that software techniques are useful for hardware debugging, we believe that the core data propagation logic of LossCheck could be generalized and adapted to other sophisticated FPGA debugging tools.

8 CONCLUSION

The proliferation of reconfigurable hardware has enabled a software-like rapid development cycle in which teams relax verification efforts. While the community has expended effort into bug finding tools (e.g., simulation-based testing tools), very little work has focused on localizing the root cause of hardware bugs. In this work, we performed a study of bugs in open-source FPGA designs and showed that hardware bugs follow a similar taxonomy to software bugs. We argue that hardware bugs are amenable to software-style hybrid static/dynamic program analysis and monitor tools and provide a toolset that aids FPGA debugging and facilitates greater confidence in emerging test-deploy-patch FPGA development cycles.

ACKNOWLEDGMENTS

We thank our shepherd, Adrian Sampson, for his guidance and the anonymous reviewers for their insightful feedback. Additionally, we thank the numerous hardware developers whose advice and feedback helped us in our efforts. This work was supported by the NSF CAREER award 1942218, the NSF DGE award 1256260, a Google fellowship, and the Applications Driving Architectures (ADA) Research Center (a JUMP Center co-sponsored by SRC and DARPA).

A ARTIFACT APPENDIX

A.1 Abstract

The artifact includes 20 hardware bugs, each of which can be reproduced with Verilator in a push-button manner. It also includes the five tools we designed to help bug localization (i.e., SignalCat, FSM Monitor, Statistics Monitor, Dependency Monitor, and LossCheck), examples of using each bug localization tool, and instructions for the figures in the paper. Below we describe each of these components in more detail:

A.2 Artifact check-list (meta-information)

- **Included Programs:** 20 reproducible hardware bugs and 5 debugging tools
- **Required Compilation tools:** Verilator, Make, C/C++, Vivado, Quartus, VCS
- **Runtime environment:** Ubuntu 20.04

A.3 Description

A.3.1 Access. The source code and tutorial are available via GitHub[2] and Zenodo[3].

A.3.2 Software dependencies. All experiments are conducted under Ubuntu 20.04. Reproducing the 20 hardware bugs requires GCC 9.3.0, G++ 9.3.0, Make 4.2.1, and a modified version of Verilator, which is included the artifact repository. Using the tools and reproducing the evaluation results requires additional software, including Vivado 2020.2, Quartus Prime Pro 17.0 (with the necessary licenses—i.e., `6AF7 00FB`, `6AF7 0119`, `6AF7 011A`, `6AF7 011B`—and platform files for Skylake HARP), and Synopsys VCS MX 2017.03.

A.4 Installation and experiment workflow

We provide detailed tutorials in the `README.md` file of the artifact repository. In the tutorial, we describe (1) how to install the repository, (2) how to reproduce each bugs in the repository, and (3) how to reproduce the evaluation result in §6.

A.5 Expected results

The following three results are expected to be reproduced:

- All bugs listed in Table 2 can be reproduced in a push-button manner. Specifically, for each bug, the user should expect an error message printed out after entering the command described in the tutorial.
- After instrumenting the hardware designs with our tools, the resource overhead reported by the synthesis tool (Quartus or Vivado) matches Figure 2.
- LossCheck reports the register where the data loss occurs for the 6 data loss bugs. For 5 bugs, LossCheck does not incur false positive. After instrumentation, the resource usage reported by Quartus matches Figure 3.

REFERENCES

[1] [n. d.]. https://github.com/efeslab/hardware-bugbase.
[2] [n. d.]. https://zipcpu.com.
[3] [n. d.]. https://github.com/ZipCPU/sdspi.
[4] [n. d.]. https://zipcpu.com/formal/2018/12/28/axilite.html.
[5] [n. d.]. https://zipcpu.com/dsp/2020/04/20/axil2axis.html.
[6] [n. d.]. https://github.com/open-sdr/openwifi-hw.
[7] [n. d.]. https://github.com/jbush001/NyuziProcessor.
[8] [n. d.]. https://github.com/openhwgroup/cva6.
[9] [n. d.]. https://github.com/SpinalHDL/VexRiscv.
[10] [n. d.]. https://github.com/progranism/Open-Source-FPGA-Bitcoin-Miner.
[11] [n. d.]. https://github.com/corundum/corundum.
[12] [n. d.]. https://github.com/alexforencich/verilog-ethernet.
[13] [n. d.]. https://github.com/analogdevicesinc/hdl.
[14] [n. d.]. https://github.com/alexforencich/verilog-axis.
[15] [n. d.]. https://github.com/mjc0608/really-simple-fadd.
[16] [n. d.]. AXI Hardware ICAP. https://www.xilinx.com/products/intellectual-property/axi_hwicap.html.

[2] https://github.com/efeslab/asplos22-hardware-debugging-artifact
[3] https://doi.org/10.5281/zenodo.5855030

[17] [n. d.]. Intel Quartus Prime Software Suite. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html.
[18] [n. d.]. Questa Verification & Simulation. https://eda.sw.siemens.com/en-US/ic/questa/simulation.
[19] [n. d.]. Vivado Design Suite. https://www.xilinx.com/products/design-tools/vivado.html.
[20] [n. d.]. Xcelium Logic Simulation. https://www.cadence.com/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html.
[21] [n. d.]. Xilinx Kintex-7 FPGA KC705 Evaluation Kit. https://www.xilinx.com/products/boards-and-kits/ek-k7-kc705-g.html.
[22] 1985. IEEE Standard for Binary Floating-Point Arithmetic. *ANSI/IEEE Std 754-1985* (1985), 1–20. https://doi.org/10.1109/IEEESTD.1985.82928
[23] 2015. https://www.exostivlabs.com/fpga-debug-flow-should-be-improved/.
[24] 2018. https://github.com/omphardcloud/hardcloud/tree/master/samples/sha512.
[25] 2018. https://github.com/omphardcloud/hardcloud/tree/master/samples/reed_solomon_decoder.
[26] 2018. https://github.com/omphardcloud/hardcloud/tree/master/samples/grayscale.
[27] 2018. https://zipcpu.com/dsp/2018/10/02/fft.html.
[28] 2018. AXI Protocol Checker v2.0. https://www.xilinx.com/support/documentation/ip_documentation/axi_protocol_checker/v2_0/pg101-axi-protocol-checker.pdf.
[29] 2018. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)* (2018), 1–1315. https://doi.org/10.1109/IEEESTD.2018.8299595
[30] 2020. IEEE Standard for Universal Verification Methodology Language Reference Manual. *IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017)* (2020), 1–458. https://doi.org/10.1109/IEEESTD.2020.9195920
[31] Alibaba. [n. d.]. Deep Dive into Alibaba Cloud F3 FPGA as a Service Instances. https://www.alibabacloud.com/blog/deep-dive-into-alibaba-cloud-f3-fpga-as-a-service-instances_594057.
[32] Altera. 2013. Implementing State Machines (Verilog HDL). https://www.intel.com/content/www/us/en/programmable/quartushelp/13.0/mergedProjects/hdl/vlog/vlog_pro_state_machines.htm.
[33] Amazon. [n. d.]. Amazon EC2 F1 Instances - Run Customizable FPGAs in the AWS Cloud. https://aws.amazon.com/ec2/instance-types/f1.
[34] Hari Angepat, Gage Eads, Christopher Craik, and Derek Chiou. 2010. NIFD: Non-intrusive FPGA Debugger–Debugging FPGA'Threads' for Rapid HW/SW Systems Prototyping. In *2010 International Conference on Field Programmable Logic and Applications*. IEEE, 356–359.
[35] ARM. 2021. AMBA AXI and ACE Protocol Specification.
[36] Osama G Attia, Tyler Johnson, Kevin Townsend, Philip Jones, and Joseph Zambreno. 2014. Cygraph: A reconfigurable architecture for parallel breadth-first search. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. IEEE, 228–235.
[37] Sameh Attia and Vaughn Betz. 2020. Feel Free to Interrupt: Safe Task Stopping to Enable FPGA Checkpointing and Context Switching. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 13, 1 (2020), 1–27.
[38] Sameh Attia and Vaughn Betz. 2020. StateMover: Combining Simulation and Hardware Execution for Efficient FPGA Debugging. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '20)*. Association for Computing Machinery, New York, NY, USA, 175–185. https://doi.org/10.1145/3373087.3375307
[39] Ivan Beschastnikh, Jenny Abrahamson, Yuriy Brun, and Michael D Ernst. 2011. Synoptic: Studying logged behavior with inferred models. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 448–451.
[40] Jayaram Bhasker. 1999. *A Vhdl Primer*. Prentice-Hall.
[41] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. 2020. hXDP: Efficient Software Packet Processing on {FPGA} NICs. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 973–990.
[42] Stuart Byma, J Gregory Steffan, Hadi Bannazadeh, Alberto Leon Garcia, and Paul Chow. 2014. Fpgas in the cloud: Booting virtualized hardware accelerators with openstack. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 109–116.
[43] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. 33–36.
[44] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D Brown, and Jason H Anderson. 2013. LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *ACM Transactions on Embedded Computing Systems (TECS)* 13, 2 (2013),

1–27.

[45] Ciro Ceissler, Ramon Nepomuceno, Marcio Pereira, and Guido Araujo. 2018. Automatic offloading of cluster accelerators. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 224–224.

[46] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. 2014. Enabling FPGAs in the cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers*. 1–10.

[47] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2021. ThunderGP: HLS-based Graph Processing Framework on FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 69–80.

[48] Young-Kyu Choi, Yuze Chi, Jie Wang, and Jason Cong. 2020. FLASH: Fast, Parallel, and Accurate Simulator for HLS. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 12 (2020), 4828–4841.

[49] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. {REPT}: Reverse debugging of failures in deployed software. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 17–32.

[50] Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Kanuparthi, Hareesh Khattri, Jason M Fung, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. 2019. Hardfails: Insights into software-exploitable hardware bugs. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 213–230.

[51] Edsger W. Dijkstra and DIJKSTRA EW. 1972. Information streams sharing a finite buffer. (1972).

[52] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure accelerated networking: Smartnics in the public cloud. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 51–66.

[53] Kermin Fleming and Michael Adler. 2016. The LEAP FPGA operating system. In *FPGAs for Software Programmers*. Springer, 245–258.

[54] Harry Foster. 2020. 2020 Wilson Research Group functional verification study: FPGA functional verification trend report.

[55] Jeffrey Goeders and Steven JE Wilton. 2014. Effective FPGA debug for high-level synthesis generated circuits. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–8.

[56] Jeffrey Goeders and Steve JE Wilton. 2015. Using dynamic signal-tracing to debug compiler-optimized HLS circuits on FPGAs. In *2015 IEEE 23rd annual international symposium on field-programmable custom computing machines*. IEEE, 127–134.

[57] Daniel Holanda Noronha, Ruizhe Zhao, Jeff Goeders, Wayne Luk, and Steven JE Wilton. 2019. On-chip fpga debug instrumentation for machine learning applications. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 110–115.

[58] 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 Transactions on Design Automation of Electronic Systems (TODAES)* 24, 1 (2018), 1–24.

[59] Eddie Hung and Steven JE Wilton. 2012. Scalable signal selection for post-silicon debug. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21, 6 (2012), 1103–1115.

[60] Intel. [n. d.]. Hardware Accelerator Research Program. https://software.intel.com/en-us/hardware-accelerator-research-program.

[61] Intel. [n. d.]. Intel High Level Synthesis Compiler. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html.

[62] Intel. 2020. Intel Quartus Prime Pro Edition User Guide: Debug Tools. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-qpp-debug.pdf.

[63] Himanshu Jain, Daniel Kroening, Natasha Sharygina, and Edmund Clarke. 2005. Word level predicate abstraction and refinement for verifying RTL verilog. In *Proceedings of the 42nd annual Design Automation Conference*. 445–450.

[64] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, et al. 2018. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 29–42.

[65] Sagar Karandikar, Albert Ou, Alon Amid, Howard Mao, Randy Katz, Borivoje Nikolić, and Krste Asanović. 2020. FirePerf: FPGA-accelerated full-system hardware/software performance profiling and co-design. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 715–731.

[66] Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. 2017. Lazy Diagnosis of In-Production Concurrency Bugs. In *SOSP*. Shanghai, China. https://doi.org/10.1145/3132747.3132767

[67] Baris Kasikci, Cristiano Pereira, Gilles Pokam, Benjamin Schubert, Malandal Musuvathi, and George Candea. 2015. Failure Sketches: A Better Way to Debug. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. USENIX Association, Kartause Ittingen, Switzerland.

[68] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. 2015. Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-Production Failures. In *SOSP*. Monterey, CA. https://doi.org/10.1145/2815400.2815412

[69] Baris Kasikci, Cristian Zamfir, and George Candea. 2013. RaceMob: Crowdsourced Data Race Detection. In *SOSP*. Farmington, PA. https://doi.org/10.1145/2517349.2522736

[70] Tanvir Ahmed Khan, Nathan Brown, Akshitha Sriraman, Niranjan K Soundararajan, Rakesh Kumar, Joseph Devietti, Sreenivas Subramoney, Gilles A Pokam, Heiner Litz, and Baris Kasikci. 2021. Twig: Profile-Guided BTB Prefetching for Data Center Applications. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[71] Tanvir Ahmed Khan, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2020. I-SPY: Context-Driven Conditional Instruction Prefetching with Coalescing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 146–159.

[72] Tanvir Ahmed Khan, Dexin Zhang, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2021. Ripple: Profile-Guided Instruction Cache Replacement for Data Center Applications. In *Proceedings of the 48th International Symposium on Computer Architecture*.

[73] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J Rossbach. 2018. Sharing, protection, and compatibility for reconfigurable fabric with amorphos. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 107–127.

[74] Alireza Khodamoradi, Kristof Denolf, and Ryan Kastner. 2021. S2N2: A FPGA Accelerator for Streaming Spiking Neural Networks. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 194–205.

[75] Donggyu Kim, Christopher Celio, Sagar Karandikar, David Biancolin, Jonathan Bachrach, and Krste Asanović. 2018. DESSERT: Debugging RTL Effectively with State Snapshotting for Error Replays across Trillions of Cycles. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 76–764.

[76] Ho Fai Ko and Nicola Nicolici. 2010. Automated trace signals selection using the RTL descriptions. In *2010 IEEE International Test Conference*. IEEE, 1–10.

[77] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. 2020. Do {OS} abstractions make sense on FPGAs?. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 991–1010.

[78] Dongup Kwon, Junehyuk Boo, Dongryeong Kim, and Jangwoo Kim. 2020. {FVM}: FPGA-assisted Virtual Device Emulation for Fast, Scalable, and Flexible Storage Virtualization. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 955–971.

[79] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. 2018. RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[80] Joshua Landgraf, Tiffany Yang, Will Lin, Christopher J Rossbach, and Eric Schkufza. 2021. Compiler-driven FPGA virtualization with SYNERGY. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 818–831.

[81] Suho Lee and Karem A Sakallah. 2014. Unbounded scalable verification based on approximate property-directed reachability and datapath abstraction. In *International Conference on Computer Aided Verification*. Springer, 849–865.

[82] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. 2006. Have things changed now? An empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*. 25–33.

[83] Shuang Liang, Shouyi Yin, Leibo Liu, Wayne Luk, and Shaojun Wei. 2018. FP-BNN: Binarized neural network on FPGA. *Neurocomputing* 275 (2018), 1072–1086.

[84] Xiao Liu and Qiang Xu. 2009. Trace signal selection for visibility enhancement in post-silicon validation. In *2009 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 1338–1343.

[85] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Xiaohe Cheng, Yanqiang Liu, Abel Mulugeta Eneyew, Zhengwei Qi, and Baris Kasikci. 2020. A hypervisor for shared-memory fpga platforms. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 827–844.

[86] Marco Antonio Merlini, Isamu Poy, and Paul Chow. 2021. Interactive Debugging at IP Block Interfaces in FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 138–144.

[87] Roberto Millón, Emmanuel Frati, and Enzo Rucci. 2020. A comparative study between HLS and HDL on SoC for image processing applications. *arXiv preprint arXiv:2012.08320* (2020).

[88] Muhsen Owaida, David Sidler, Kaan Kara, and Gustavo Alonso. 2017. Centaur: A framework for hybrid CPU-FPGA databases. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 211–218.

[89] Clément Pit-Claudel, Thomas Bourgeat, Stella Lau, and Adam Chlipala. 2021. Effective simulation and debugging for a high-level hardware language using software compilers. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 789–803.

[90] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 13–24.

[91] Hao Qian and Yangdong Deng. 2011. Accelerating RTL simulation with GPUs. In *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 687–693.

[92] Weikang Qiao, Jieqiong Du, Zhenman Fang, Michael Lo, Mau-Chung Frank Chang, and Jason Cong. 2018. High-throughput lossless compression on tightly coupled CPU-FPGA platforms. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 37–44.

[93] Alastair Reid. 2016. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In *2016 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 161–168.

[94] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. 2016. End-to-end verification of processors with ISA-Formal. In *International Conference on Computer Aided Verification*. Springer, 42–58.

[95] Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram Adve. 2013. Using likely invariants for automated software fault localization. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*. 139–152.

[96] Sahand Salamat, Armin Haj Aboutalebi, Behnam Khaleghi, Joo Hwan Lee, Yang Seok Ki, and Tajana Rosing. 2021. NASCENT: Near-Storage Acceleration of Database Sort on SmartSSD. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 262–272.

[97] Eric Schkufza, Michael Wei, and Christopher J Rossbach. 2019. Just-in-time compilation for Verilog: A new technique for improving the FPGA programming experience. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 271–286.

[98] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, USA, 28.

[99] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications*. 62–71.

[100] Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATEC '05)*. USENIX Association, USA, 2.

[101] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–12.

[102] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–12.

[103] David Sidler and Ken Eguro. 2016. Debugging framework for FPGA-based soft processors. In *2016 International Conference on Field-Programmable Technology (FPT)*. IEEE, 165–168.

[104] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. 2017. Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 403–415.

[105] Wilson Snyder. 2021. https://www.veripool.org/verilator/.

[106] Hayden Kwok-Hay So and Robert W Brodersen. 2007. *Borph: An operating system for fpga-based reconfigurable computers*. Citeseer.

[107] Synopsys. 2021. VCS Functional Verification Solution. https://www.synopsys.com/verification/simulation/vcs.html.

[108] Shinya Takamaeda-Yamazaki. 2015. Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL. In *Applied Reconfigurable Computing (Lecture Notes in Computer Science, Vol. 9040)*. Springer International Publishing, 451–460. https://doi.org/10.1007/978-3-319-16214-0_42

[109] Donald Thomas and Philip Moorby. 2008. *The Verilog® Hardware Description Language*. Springer Science & Business Media.

[110] Timothy Trippel, Kang G Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. 2021. Fuzzing Hardware Like Software. *arXiv preprint arXiv:2102.02308* (2021).

[111] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. 2017. P4fpga: A rapid prototyping framework for p4. In *Proceedings of the Symposium on SDN Research*. 122–135.

[112] Qinggang Wang, Long Zheng, Yu Huang, Pengcheng Yao, Chuangyi Gui, Xiaofei Liao, Hai Jin, Wenbin Jiang, and Fubing Mao. 2021. GraSU: A Fast Graph Update Library for FPGA-based Dynamic Graph Processing. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 149–159.

[113] Wei Wang, Miodrag Bolic, and Jonathan Parri. 2013. pvFPGA: Accessing an FPGA-based hardware accelerator in a paravirtualized environment. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE, 1–9.

[114] Stephen Williams. [n. d.]. Icarus Verilog. http://iverilog.icarus.com/.

[115] Clifford Wolf. 2016. Yosys open synthesis suite.

[116] Xilinx. [n. d.]. SDAccel Development Environment. https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.

[117] Xilinx. [n. d.]. Vitis High-Level Synthesis. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[118] Xilinx. 2015. Finite State Machines. https://www.xilinx.com/support/documentation/university/Vivado-Teaching/HDL-Design/2015x/VHDL/docs-pdf/lab10.pdf.

[119] Xilinx. 2016. Integrated Logic Analyzer v6.2. https://www.xilinx.com/support/documentation/ip_documentation/ila/v6_2/pg172-ila.pdf.

[120] Hangchen Yu, Arthur Michener Peters, Amogh Akshintala, and Christopher J Rossbach. 2020. AvA: Accelerated Virtualization of Accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 807–825.

[121] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. 2012. Characterizing logging practices in open-source software. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 102–112.

[122] H. Zeng, C. Zhang, and V. Prasanna. 2017. Fast Generation of High Throughput Customized Deep Learning Accelerators on FPGAs. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 1–8. https://doi.org/10.1109/RECONFIG.2017.8279792

[123] Yue Zha and Jing Li. 2020. Virtualizing FPGAs in the cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 845–858.

[124] Yue Zha and Jing Li. 2021. When application-specific ISA meets FPGAs: a multi-layer virtualization framework for heterogeneous cloud FPGAs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 123–134.

[125] Min Zhang, Linpeng Li, Hai Wang, Yan Liu, Hongbo Qin, and Wei Zhao. 2019. Optimized compression for implementing convolutional neural networks on fpga. *Electronics* 8, 3 (2019), 295.

[126] 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 51st Annual IEEE/ACM International Symposium on Microarchitecture* (Fukuoka, Japan) *(MICRO-51)*. IEEE Press, 815–827. https://doi.org/10.1109/MICRO.2018.00071

[127] Yichi Zhang, Junhao Pan, Xinheng Liu, Hongzheng Chen, Deming Chen, and Zhiru Zhang. 2021. FracBNN: Accurate and FPGA-Efficient Binary Neural Networks with Fractional Activations. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 171–182.

[128] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. 2017. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 15–24.

[129] Shijie Zhou and Viktor K Prasanna. 2017. Accelerating graph analytics on CPU-FPGA heterogeneous platform. In *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 137–144.

[130] Gefei Zuo, Jiacheng Ma, Andrew Quinn, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. 2021. Execution Reconstruction: Harnessing Failure Reoccurrences for Failure Reproduction. In *ACM SIGPLAN conference on Programming language design and implementation*.