DirectFuzz: Automated Test Generation for RTL Designs using Directed Graybox Fuzzing

Sadullah Canakci[1], Leila Delshadtehrani[1], Furkan Eris[1], Michael Bedford Taylor[2], Manuel Egele[1], and Ajay Joshi[1]
[1]Department of ECE, Boston University
[2]Department of ECE and Paul G. Allen School of CSE, University of Washington
{scanakci, delshad, fe, megele, joshi}@bu.edu, {prof.taylor}@gmail.com

***Abstract*— A critical challenge in RTL verification is to generate effective test inputs. Recently, RFUZZ proposed to use an automated software testing technique, namely Graybox Fuzzing, to effectively generate test inputs to maximize the coverage of the whole hardware design. For a scenario where a tiny fraction of a large hardware design needs to be tested, the RFUZZ approach is extremely time consuming. In this work, we present DirectFuzz, a directed test generation mechanism. DirectFuzz uses Directed Graybox Fuzzing to generate test inputs targeted towards a module instance, which enables targeted testing. Our experimental results show that DirectFuzz covers the target sites up to 17.5× faster (2.23× on average) than RFUZZ on a variety of RTL designs.**

***Index Terms*—Graybox fuzzing, RTL verification, coverage directed test generation, RISC-V**

I. Introduction

A critical challenge in the RTL verification process is to generate test inputs that can cover all parts of the RTL design, and identify discrepancies between the RTL design of a hardware system and its functional specifications. Although several commercial tools and studies based on using formal verification techniques for test generation [7], [9], [19] have shown promising results in the verification of RTL designs, test generation using formal methods usually suffers from a state explosion problem as the size of the design increases [8], [10]. Therefore, researchers have proposed several dynamic verification techniques that rely on RTL simulators [2]. In an ideal scenario, a dynamic verification technique should generate inputs that cover every single circuit component in the design.

Unfortunately, it is challenging and time-consuming to achieve high coverage with manually crafted inputs, especially for large designs with millions of gates. Therefore, researchers proposed several coverage guided test generation (CDG) mechanisms [12], [21], [25]. These mechanisms obtain coverage feedback from the Design Under Test (DUT) to automatically fine-tune the biases of test generators, for example tuning the parameters of a Bayesian network used for test generation [12]. However, these mechanisms are usually either DUT-specific or require in-depth design knowledge for the initial setup. To increase the general applicability of a test generation mechanism, a recent work named RFUZZ [16] leveraged Graybox Fuzzing (GF), which is widely used for testing software applications. GF is an automated input generation technique at its core, and so it could also be used to test hardware designs. In Figure 1, we summarize how GF is utilized as a hardware test generation mechanism. The GF-based test generation mechanism records the achieved coverage for each test input obtained from an input queue using RTL simulations. It generates new inputs from 'interesting' inputs (i.e., the inputs that increase coverage) by employing a variety of mutations. GF requires very limited domain knowledge and manual effort for constructing the testing environment.

While RFUZZ is promising, it is not completely suitable for hardware design as hardware design is an incremental process, where new components are gradually added instead of designing the entire system in one step. After adding a new hardware component, not all of the older verified components of the DUT need to undergo a thorough testing. Consider a scenario where an existing processor design is extended with a new feature such as a floating point unit or a scenario where we replace the existing branch predictor in the processor with a better branch predictor. For these scenarios, the test-time budget needs to be allocated for the verification of the new/modified components and their interactions with the rest of the DUT. Unfortunately, RFUZZ is agnostic of such scenarios and it focuses on the whole processor design. As a result, it spends unnecessary effort to maximize coverage of the **whole** design rather than a specific **part** of the hardware design.

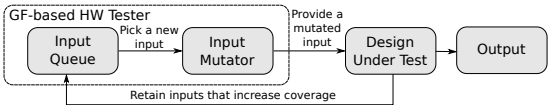

Fig. 1. A simplified overview of a graybox fuzzer.

In this work, we present DirectFuzz, which generates test inputs to maximize the coverage of a specific part of the hardware design. At a high-level, DirectFuzz spends most of its time budget on reaching specific target sites instead of covering the whole design. Here, target sites refer to module instances that need to be tested. DirectFuzz achieves this goal by leveraging Directed Graybox Fuzzing (DGF) [6], an approach used by the software community for patch testing, and identifying special types of bugs (e.g., use-after-free). DirectFuzz differs from RFUZZ in two ways. First, RFUZZ selects the test inputs from the input queue in the order they are inserted. This increases the total time for reaching target sites when an effective test input (the one that increases the target site coverage) resides close to the rear of a long queue. DirectFuzz prioritizes those inputs that cover at least part of the target sites when choosing the next input for the DUT in order to quickly reach target sites. Second, RFUZZ and DirectFuzz differ in the number of mutations that they apply to an input. For each test input, RFUZZ applies the same number of mutations while DirectFuzz adjusts the number of mutations based on how close the input is to the target sites. The closeness of the test input to the target site is determined by a distance metric that accounts for the module instance hierarchy. DirectFuzz also generates more inputs from a test input if it achieves high coverage in the module instances that are close to the target module instance.

Overall, the main contributions of this work are as follows:

- To the best of our knowledge, we are the first to apply the notion of DGF to hardware test generation for effectively and efficiently verifying specific target sites in hardware designs.
- For generating new test inputs, we propose to use a distance metric that accounts for the instance hierarchy of a hardware design. The distance metric determines the importance of each test input.
- We demonstrate the efficacy and utility of DirectFuzz on several real-world RTL designs. DirectFuzz covers the same set of target sites up to 17.5× (on average 2.23×) faster than RFUZZ under the same time budget. In the spirit of open science, we will make DirectFuzz publicly available.

978-1-6654-3274-0/21/$31.00 ©2021 IEEE

II. Background

In this section, we provide the basics of GF and explain how RFUZZ applies GF to the test input generation problem for hardware.

A. Graybox Fuzzing

Fuzzing is the process of executing a Program Under Test (PUT) repeatedly with generated inputs (seeds) to trigger software bugs. Due to its simplicity, practicality, and capability of discovering bugs, fuzzing has become very popular in the last decade in the software community. Several software companies such as Google and Microsoft have created their own fuzzing platforms [5], [18] and identified a plethora of bugs in popular programs. Most modern fuzzers primarily focus on GF, which relies on dynamic information about the execution of the PUT, such as code coverage feedback. A graybox fuzzer consists of the following stages as detailed in Algorithm 1: **(S1)** Provide the fuzzer with an initial seed corpus and total fuzzing duration; **(S2)** Pick a seed from the seed corpus; **(S3)** Assign an energy level to the current seed to determine how many new seeds should be generated from that particular seed; **(S4)** Mutate the seed to generate N new seeds, where N is determined by the seed energy; **(S5)** Execute the PUT with the seed to produce an observation; and **(S6)** Analyze this observation to determine if the seed is "interesting" enough (i.e., increases coverage) for further mutation or causes a crash.

B. RFUZZ

DirectFuzz is closely related to RFUZZ [16]. In this subsection, we explain how RFUZZ adopts GF into a test generation mechanism for hardware designs. RFUZZ consists of two components: a fuzzing logic, and an instrumentation suite:

The fuzzing logic requires an initial seed corpus that consists of a set of test inputs **(S1)**. An RTL design requires a rigid test input size determined by the RTL's input port widths as opposed to a software program that usually accepts an arbitrary-sized test inputs. RFUZZ generates a bit vector of size N, where N is determined by the input port widths and total number of test cycles. Next, the fuzzing logic chooses a test input from a queue in FIFO order **(S2)** and assigns an energy level **(S3)** which determines the number of mutations that need to be applied in **(S4)**. Note that RFUZZ uses the same energy level for each test input, thereby performing the same number of mutations. Similar to graybox fuzzers targeting software programs, RFUZZ implements several deterministic (e.g., a single bit flip at a constant offset) and non-deterministic mutations (e.g., random byte overwrite). Finally, the fuzzing logic retains any test input that increases the coverage of the RTL design **(S5-S6)**.

The instrumentation suite is in charge of collecting coverage feedback when the current input exercises the DUT. In software testing, graybox fuzzers mostly rely on covered branch instructions (e.g., `jump`). The branches (like if-then-else control structure or switch control structure) written in a Hardware Description Language (HDL) for an RTL design are mapped to multiplexers in the circuit. Also, there are usually multiple activated multiplexers during any cycle in hardware as opposed to a sequential software program that can only trigger one branch instruction at a time. RFUZZ takes into account these two fundamental differences in its definition of coverage metric. Specifically, RFUZZ uses *mux control coverage* metric which considers the select signal of each 2:1 multiplexer as a coverage point[1]. It instruments the DUT with additional bookkeeping logic for each multiplexer to observe the value of the selection signal when exercising the DUT with an input. The coverage feedback includes the multiplexers whose selection bits are toggled. RFUZZ defines the achieved coverage in a design as the ratio of the number of multiplexers with selection bits toggled over total number of multiplexer selection signals in the RTL.

DirectFuzz modifies the second and third stages of Algorithm 1 to convert a graybox fuzzer into a DGF. DirectFuzz implements an input selection scheme that gives priority to inputs that are likely to increase coverage of the target sites **(S2)**. Moreover, DirectFuzz implements a power scheduling algorithm that assigns different energy levels to the inputs **(S3)**, which leads to different number of employed mutations on each input. These modifications enable DirectFuzz to generate test inputs tailored for specific targets in the hardware design. We provide the details of our modifications in Section IV-C.

Algorithm 1: Graybox Fuzzing

```
(S1) Input : Initial Seed Corpus S, time_limit
Output: Crashing Inputs C
C ← ∅;
while time_elapsed < time_limit do
    /* Our modifications are highlighted */
    (S2) s ← ChooseNext(S);
    (S3) e ← AssignEnergy(s);
    for i = 1 to e do
        (S4) m' = MutateInput(s);
        (S5) o = ExecuteDUT(m');
        (S6) if o == IS_CRASHING then
            add m' to C;
        else if o == IS_INTERESTING then
            add m' to S;
    end
end
return C
```

III. Related Work

Coverage Directed Test Generation (CDG) is a widely-used technique for the verification of RTL designs. In this technique, the constraints of a test generator are automatically driven by the coverage feedback so that the test input generated in the next round can increase the overall coverage. For instance, MicroGP [21] aims to verify the whole microprocessor design by generating test inputs using an instruction template based on genetic programming. The fitness value that drives the searching of an instruction sequence is determined by the statement coverage. Fine et al. [12] propose a CDG mechanism based on Bayesian networks. Unfortunately, setting up the network is not a straightforward task and requires in-depth expertise in the design specifications of the RTL design. Wagner et al. [25] presented a CDG framework that uses a Markov chain model. The weights of the model are fine-tuned based on the collected coverage. This framework relies on the abstract form of the DUT that needs to be crafted manually in the form a custom template. Therefore, it requires deep domain knowledge. Overall, CDG mechanisms aim to find a balance between the amount of domain knowledge applied to the framework and general applicability of the mechanism [13].

Fuzzing-based approaches like RFUZZ [16] and DirectFuzz minimize the amount of domain knowledge for their testing mechanisms. For some RTL designs, this design choice may lead to lower coverage compared to a fine-tuned CDG tool designed with expert feedback [12], [21], [25]. However, the fuzzing-based approaches provide a more generic framework that can be used for a variety of RTL designs without requiring additional integration effort.

Several works utilize formal methods to generate test inputs for hardware designs [7], [9], [19]. Formal methods that rely on symbolic execution have a well-known state explosion problem due to the exponentially growing execution paths in the RTL designs [8], [10]. To mitigate this problem, researchers in the hardware community leveraged a software testing technique, called concolic testing, which

[1]RFUZZ converts any other multiplexer type such as a 4:1 multiplexer into a set of 2:1 multiplexers.

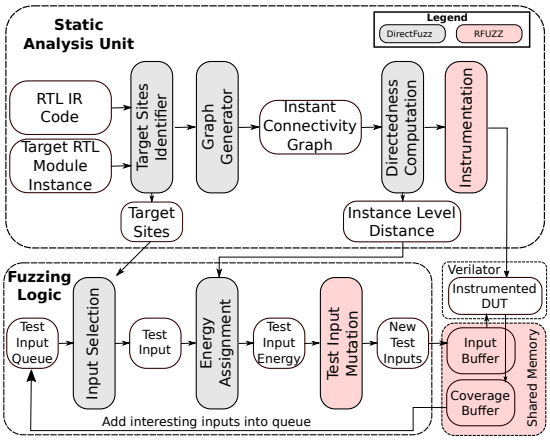

Fig. 2. Overview of DirectFuzz. The gray boxes represent the components of DirectFuzz. The red boxes correspond to the components that DirectFuzz leverages from RFUZZ.

interleaves concrete execution with symbolic execution. QUEBS [4] uses heuristics to efficiently select the execution path, collects the path constraints, and provides an SMT solver with these constraints to generate a test input. While QUEBS aims to maximize overall branch coverage, a follow-up work by the same authors [3] proposed a directed test generation mechanism tailored for a single target in the RTL design. Lyu et al. [17] extended this work to support the test generation for multiple targets to minimize the number of overlapping searches. These three approaches are computationally less expensive than their symbolic execution based counterparts.

Concolic test mechanisms rely on constraint solvers (as opposed to graybox fuzzers) to generate a test input, which hinder the scalability of the techniques [15]. At the same time, graybox fuzzers may spend enormous time to find inputs for some of the branches in the RTL, which can be trivially found by a concolic tester. Recent works in software testing [22], [26] propose hybrid systems where they use fuzzing and concolic testing alternating during program testing. Similar hybrid systems could potentially be utilized for implementing effective hardware test input generation mechanisms but are out of scope for our work.

IV. Directed Test Generation For Hardware

In this section, we present DirectFuzz, our proposed DGF technique for hardware verification.

A. Overview

In Figure 2, we provide the overview of DirectFuzz, which consists of two components, **(1)** The Static Analysis Unit and **(2)** the Fuzzing Logic. The Static Analysis Unit takes the Intermediate Representation (IR) of the RTL design [14] as an input and applies several IR passes to extract information that the Fuzzing Logic will use to generate the test inputs. Specifically, the Static Analysis Unit has three main tasks: 1) Identify the coverage points in the target module instance; 2) Generate an instance connectivity graph to calculate the relative distance of each module instance with respect to the target module instance; 3) Create an instrumented DUT that includes the necessary bookkeeping logic to record coverage for each test input.

The Fuzzing Logic is in charge of choosing the next test input, assigning an energy value to the test input based on a power scheduling function, and performing input mutations. The instrumented DUT and the Fuzzing Logic communicate via a shared memory region allocated by the operating system to exchange generated inputs and the achieved coverage information per input.

B. Static Analysis Unit

1) Target Module Instance Selection: DirectFuzz aims to generate test inputs towards maximizing the coverage of specific target sites in an RTL design. More specifically, the target sites refer to the multiplexer selection signals that reside in a specific **module instance** chosen by a verification engineer. Note that DirectFuzz accepts a module instance as a target point rather than a module. When there are multiple module instances produced by the same module, the verification engineer needs to account for the position of each module instance with respect to the DUT to determine the target module instance.

The verification engineer can determine the target module instance with a manual or an automated process. For the former, she can choose the name of the module instance if she is directly aware of a change in the RTL code of a specific module instance. For the latter, she can determine the target module instance with software tools (e.g., git-diff and svn diff) and extract the modified instances between two versions of an RTL code.

2) Target Sites Identifier: As detailed in Section II-B, the coverage points for DirectFuzz are multiplexer selection signals. To effectively generate test-cases for the target module instance, it is necessary to identify which multiplexer selection signals are covered by the current test input. First, the Target Sites Identifier (TSI) analyzes the provided RTL IR code and extracts all multiplexer selection signals in the whole RTL design. Next, TSI labels any multiplexer selection signal as 'target' if the multiplexer selection signal is part of the provided target module instance. The identified multiplexer selection signals are provided to the fuzzing logic.

3) Module Instance Connectivity Graph Generation: An important part of the directed test generation is to calculate the distance of each module instance in the RTL design with respect to the target module instance. In fact, if the target module instance constitutes a small portion of the RTL design, test inputs might mostly cover the non-target multiplexer selection signals during fuzzing. Therefore, a module instance hierarchy is essential to realize which parts of the DUT are covered by the current test input and how close the covered sites are to the target sites.

The graph generator creates a module instance hierarchy of the RTL design in the form of a directed module instance connectivity graph by using the RTL IR code. In this graph, the nodes represent module instances in the RTL design while the edges represent the module instance connections. Two module instances are considered as connected if either instance is a subinstance ('child' instance) of the other instance or they are subinstances with the same 'parent' instance. The generated graph is a directed graph for two reasons: 1) it presents the hierarchical view of the overall hardware design with parent instances and child instance(s); and 2) it presents the communication direction between two module instances. For example, if instance A provides data to the input ports of instance B but not vice versa, the direction of the edge should be only from A to B.

To understand the process of graph generation, we use an example 1-Stage processor (Sodor [1]) with its corresponding module instance connectivity graph (see Figure 3). The processor consists of seven module instances – `proc`, `mem`, `core`, `c`, `d`, `async_data`, `csr`, one instance of each of the `Sodor1Stage`, `Memory`, `Core`, `CtlPath`, `DatPath`, `AsyncReadMem`, `CSRFile` modules, respectively. As shown in Figure 3 (on the right), any connection between a child instance and its parent instance is represented with a one-way edge (e.g., from `proc` to `mem` and from `proc` to `core`). We use directed edges to represent the connections between the child instances (for example, `c` and `d`).

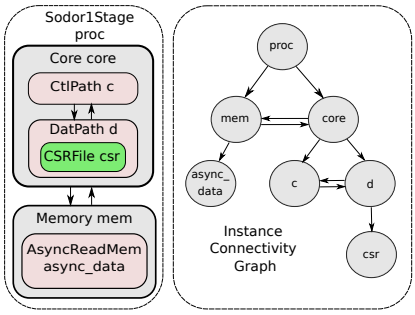

Fig. 3. RTL design (on the left) and the corresponding module instance connectivity graph (on the right) of Sodor 1-Stage processor.

4) Directedness Computation: DirectFuzz calculates the instance level distance of the RTL design to determine if a test input covers multiplexer selection signals that are closer to the target sites. Our intuition is that mutating an input that covers multiplexer selection signals from the instances that directly interact with the target instance is likely to increase coverage in the target instance. For example, if the target site is the `csr` instance in Figure 3, an input that covers multiplexer selection signals in `d` is more likely to result in an increase in the multiplexer selection signal coverage of our target `csr` given that `d` and `csr` are directly connected to each other. Based on this intuition, we define a metric called instance level distance d_{il} for multiplexer selection signal m with respect to target instance I_t as

$$d_{il}(m, I_t) = \begin{cases} undefined & \text{if } S(I_t, I_m) = \emptyset \\ S(I_t, I_m) & \text{otherwise} \end{cases} \quad (1)$$

where I_m is the module instance in which m resides and $S(I_t, I_m)$ is the number of edges along the shortest path between I_m and I_t in the instance connectivity graph. Note that all the multiplexer selection signals that reside in the same module instance are assigned the same instance level distance. The instance level distance of a multiplexer selection signal is undefined if its corresponding module instance cannot reach the target instance. The multiplexer selection signals in the target instance are assigned to zero instance level distance. The instance level distances are used by the Fuzzing Logic to calculate the energy of a test input. We provide the details in Section IV-C2.

C. Fuzzing Logic

The Fuzzing Logic is in charge of selecting the current input, assigning energy to the selected input, and performing mutations on the selected input to generate new test inputs. As explained in Section IV-A, we use the same test input mutation mechanism implemented by RFUZZ. However, we modify the input selection mechanism and add an energy assignment mechanism in the Fuzzing Logic (see Figure 2) to adapt the notion of DGF to the hardware test generation problem. We provide the details of our changes below.

1) Input Prioritization: RFUZZ strictly chooses the next input from the input queue, which has a FIFO ordering. Unfortunately, this is not ideal when targeting a specific instance in the RTL design. Not all inputs should have the same priority because only a subset of the generated inputs increase the coverage in the target module instance. Moreover, if an input leads to a new coverage point in the target instance, it is likely that the mutated inputs from that specific input will lead to new coverage points in the target instance. Therefore, the inputs should be stored in a priority queue to assign higher priorities to the inputs that have higher chance of increasing coverage in the target instance. DirectFuzz implements a separate priority queue to separately store the test inputs that covered at least one multiplexer selection signal in the target module instance. Inputs in this priority queue are always picked (in FIFO order) before picking any inputs from the regular queue. If the priority queue is empty, DirectFuzz uses an input from the regular queue in FIFO order.

2) Power Scheduling: Power scheduling determines the number of mutations that need to be applied to the current input. During the test input generation, we apply power scheduling on a selected input based on a dynamically-computed *input distance*, which is the distance of an input i to the target instance I_t. The intuition is that if the current input covers multiplexer selection signals closer to the target site, more mutations on this input should help in increasing coverage in the target instance. Below we present the formal definition of the input distance metric, and a power scheduling function that relies on the input distance.

The input distance $d(i, I_t)$ is computed as

$$d(i, I_t) = \frac{\sum_{m \in C(i)} d_{il}(m, I_t)}{|C(i)|} \quad (2)$$

where $C(i)$ is the set of all multiplexer selection signals that the input i covered in the RTL design. Note that $d_{il}(m, I_t)$ is defined for all $m \in C(i)$. The range of $d(i, I_t)$ is [0, d_{max}], where d_{max} represents the distance between the I_t and the instance with the largest 'shortest path' to I_t. If the input covers only the multiplexer selection signals in the instance farthest from I_t, the input distance will be d_{max}. If the input covers multiplexer selection signals only from the I_t, the input distance will be assigned zero.

The power scheduling function is defined as

$$p(i, I_t) = maxE - \left((maxE - minE) \cdot \frac{d(i, I_t)}{d_{max}} \right) \quad (3)$$

where I_t represents the target instance. $minE$ and $maxE$ are the constant lower and upper energy limits. When the $d(i, I_t)$ is zero and d_{max}, the assigned energy will be $maxE$ and $minE$, respectively. Overall, the power scheduling function favors the inputs with lower distances, thereby performing more input mutations on those inputs.

DirectFuzz assigns a power coefficient to each input i based on the power scheduling function. We calculate the input energy by multiplying the power coefficient of an input with the default mutation number provided by RFUZZ. The input energy determines the total mutation number for the input (e in Algorithm 1). In essence, DirectFuzz adjusts the total number of mutations employed by each mutator in RFUZZ based on the computed power coefficient of each input. For example, if the current mutator performs N random bit flips in RFUZZ, the same mutator performs $N \times p$ random bit flips in DirectFuzz. By adjusting the power coefficient of inputs, DirectFuzz can prevent excessive mutations that do not increase the coverage in the target site.

3) Random Input Scheduling: Our power scheduling function always favors the inputs with lower distances. Unfortunately, this greedy approach may get stuck in a local minimum instead of reaching a global minimum, thereby not covering the target multiplexer selection signals. For example, when the target module instance is `csr` for the RTL design in Figure 3, DirectFuzz can favor the inputs that increase coverage in `c` by considering multiplexer selection signals in `c` as global minimum. To prevent DirectFuzz from getting stuck at local minimums, we randomly pick an input with low energy value after an interval and schedule this input with its default energy value (i.e., `p` is set to 1). The interval of random input scheduling is determined by the coverage progress in the target instance. DirectFuzz runs the random input scheduling mechanism if there is no coverage increase in the target module instance for the scheduled last ten inputs.

TABLE I
THE EXPERIMENTAL RESULTS OF RFUZZ AND DIRECTFUZZ ON 12 MODULE INSTANCES FROM 8 RTL DESIGNS.

Benchmark	Total # of Instances	Target Instance	Total # of Mux Selection Signals	Target Instance Cell Percentage	RFUZZ		DirectFuzz		Speedup
					Coverage	Time(s)	Coverage	Time(s)	
UART	7	Tx	6	5.1%	100%	7.35	100%	0.42	17.5
		Rx	9	6.9%	88.89%	4.95	88.89%	1.71	2.89
SPI	7	SPIFIFO	5	34.4%	100%	55.84	100%	31.75	1.76
PWM	3	PWM	14	26.9%	100%	12.79	100%	100%	5.87
FFT	3	DirectFFT	107	87%	13%	0.075	13%	0.073	1.03
I2C	2	TLI2C	65	31%	98%	13.73	98%	8.49	1.61
Sodor1Stage	8	CSR	93	16.6%	96.77%	500.56	96.77%	463.63	1.08
		CtlPath	68	0.3%	100%	694.42	100%	526.53	1.32
Sodor3Stage	10	CSR	90	16.4%	98.89%	568.05	98.89%	446.29	1.27
		CtlPath	66	0.3%	100%	1283.4	100%	1034.86	1.24
Sodor5Stage	7	CSR	93	3.1%	96.77%	817.58	96.77%	322.19	2.54
		CtlPath	70	0.1%	100%	1227.35	100%	393.15	3.12
Geo. Mean	5	-	37	5.43%	82.87%	152	82.87%	29	2.23

V. EVALUATION

We implemented DirectFuzz by extending the open-source RFUZZ repository [11]. For a fair head-to-head comparison between RFUZZ and DirectFuzz, we used all the RTL designs evaluated by RFUZZ [16]. These RTL designs include several peripheral IPs (e.g., SPI [20], I2C [20], UART [20]), a DSP block (FFT) [24], a Pulse Width Modulator (PWM) [20], and three (1-stage, 3-stage, 5-stage) in-order 32-bit RISC-V processors [1]. We used FIRRTL [14] as the intermediate representation for the RTL designs. Both RFUZZ and DirectFuzz are compatible with any design expressed in FIRRTL form. We applied several FIRRTL passes to the RTL IR code in order to identify target sites, generate instance connectivity graph, and perform directedness computation. In order to collect coverage feedback, we used the FIRRTL passes provided by RFUZZ [11].

We conducted the experiments using Verilator [2] on an Intel® Core™ i7-9700 3 GHz machine. We ran each experiment for 24 hours. If all multiplexer selection signals in the target instance were covered in less than 24 hours, we terminated those experiments early. Due to the probabilistic nature of fuzzing, we repeated each experiment ten times and report the geometric mean of the ten runs for each design in Table I. To demonstrate the variation across ten runs, we provide the box (25%ile) and whisker (75%ile) plot for each design in Figure 4.

A. Evaluation Dataset and Target Module Instance Selection

We determined the target module instances from small designs (UART, SPI, PWM. I2C and FFT) based on the number of multiplexer selection signals. For these designs, we determine the module instances with the highest number of multiplexer selection signals as targets since any change in these RTL designs will likely modify these module instances. To determine the target module instances for RISC-V processors (i.e., Sodor1Stage, Sodor3Stage, and Sodor5Stage), we extracted the area of each module instance in three processor designs. We picked two module instances from the processor cores – one with the smallest area, `CtlPath` and one with largest area, `CSR`, as targets to show the impact of the target instance size on the performance of DirectFuzz. We report the cell percentage of `CtlPath` and `CSR` instances under the 'Target Instance Cell Percentage' column in Table I. For our area estimation, we synthesized the benchmark circuits using Synopsys Design Compiler [23] for GlobalFoundries 22nm process at 1 GHz clock frequency.

B. Results

In this subsection, we show the efficiency of DirectFuzz over RFUZZ where the goal is to reduce the total testing time for generating test inputs that cover a specific set of mux selection signals in a target module instance. We present the experimental results in Table I. The first column lists the design name, while the second column shows the total number of module instances in that design. We provide the name of the target instance(s) within the design in the third column with their corresponding total number of multiplexer selection signals listed in the fourth column. The achieved coverage ratio in each target instance for RFUZZ and DirectFuzz are presented in seventh and ninth column, respectively. The total time required for achieving each coverage ratio is provided under sixth and eighth columns for RFUZZ and DirectFuzz, respectively. On average, DirectFuzz covers the multiplexer selection points 2.23× faster than RFUZZ for the target instances in Table I thanks to its directed test generation mechanism.

DirectFuzz achieves the highest performance improvement (17.5×) in UART when the target module instance is `Tx`. DirectFuzz achieves the lowest performance improvement (1.08×) in Sodor1Stage RISC-V processor when the target is `CSR`. Here, the difference in the total number of mux selection signals is the key reason behind the speedup difference. It is intuitive that covering a low number of target sites in a small design (e.g., UART) is easier than covering a high number of target sites in a relatively more complex design (e.g., a processor like Sodor5Stage). However, even a small speedup in a complex design could save a lot of testing time. For example, DirectFuzz saves 834 seconds of testing time to cover all the target points in Sodor5Stage (`CtlPath`). Intuitively, DirectFuzz could reduce the hardware testing time even more for complex SoCs. We provide our hypothesis related to testing DirectFuzz on a complex SoC in Section VI.

To assess the effectiveness of DirectFuzz on different sizes of target instances, we pick two target module instances with varying target instance cell percentages for each one of the three RISC-V processors. Overall, we did not observe a direct correlation between the target module instance cell percentage and the speedup to cover the same of target points. On the one hand, DirectFuzz covers all targets in `CtlPath` faster than RFUZZ (up to 3.12×) for all three RISC-V processors even though this instance occupies a small area (0.1%) of the overall processor design. On the other hand, DirectFuzz achieves 1.27× speedup to cover the same set of target points on `CSR` target of Sodor3Stage, which has relatively high cell percentage (e.g., 16.4%).

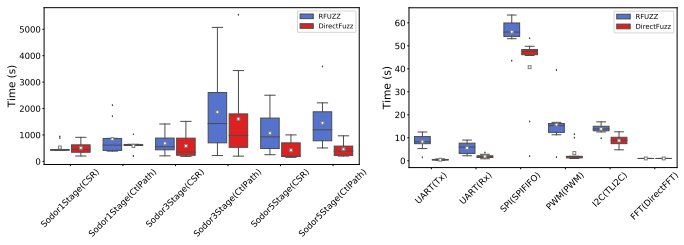

Fig. 4. Whisker plots comparing DirectFuzz and RFUZZ for various RTL designs.

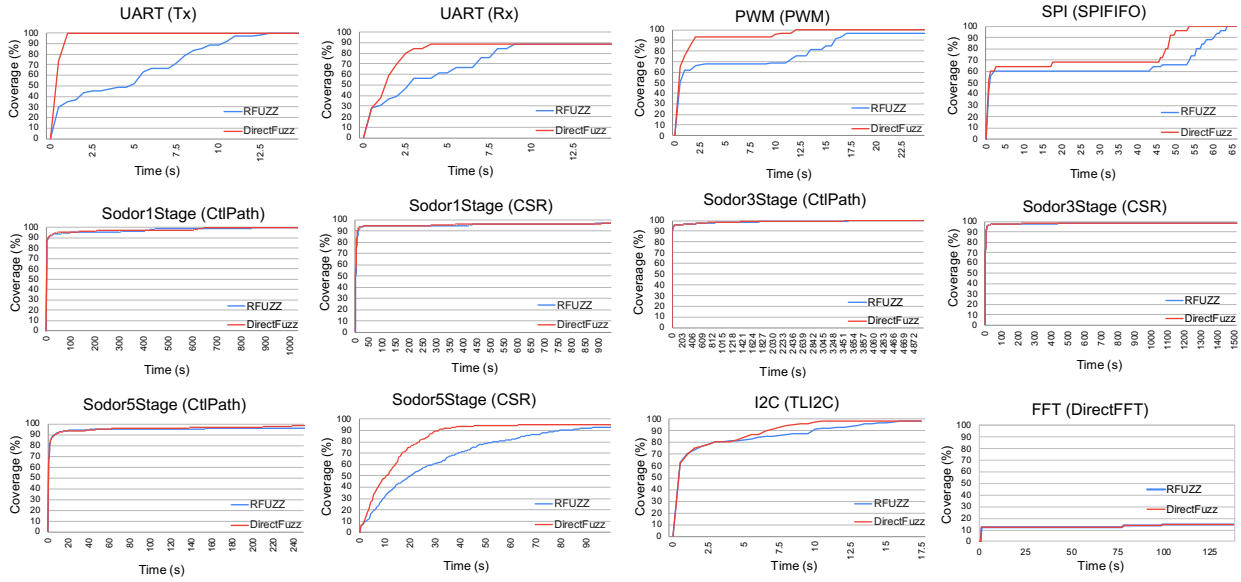

Fig. 5. Coverage progress of RFUZZ and DirectFuzz over time.

We report the coverage progress of RFUZZ and DirectFuzz over time for all the designs in Figure 5. The coverage progress is averaged over ten runs. The target instance for each design is provided inside the parenthesis of figure titles. For several benchmarks (such as UART, PWM, and Sodor5Stage for CSR target), the benefit of DirectFuzz over RFUZZ is clear based on the coverage progress. For example, DirectFuzz reaches the peak coverage more rapidly than RFUZZ for the UART benchmark (for both `Tx` and `Rx` target instances). RFUZZ's coverage gradually increases and reaches the peak coverage later than DirectFuzz. For some benchmarks (such as Sodor1Stage, Sodor3Stage, and Sodor5Stage for `CtlPath`), we observe that both RFUZZ and DirectFuzz follow a similar pace throughout the fuzzing.

VI. Future Work & Conclusion

This work presents DirectFuzz, a mechanism that generates test inputs for accelerating the testing of specific module instances in an RTL design. As opposed to prior work (RFUZZ) that aims to maximize the coverage of the RTL design, DirectFuzz aims to cover a set of target sites in a specific module instance in the RTL design. A head-to-head comparison of DirectFuzz and RFUZZ using a variety of benchmarks demonstrates that DirectFuzz achieves the same coverage on specific target sites, on average, 2.23× faster than RFUZZ.

As part of future work, we plan to enhance DirectFuzz by providing the capability to perform domain-aware but microarchitecture-agnostic mutations of inputs in the Fuzzing Logic. For example, in case of processors, one can use Instruction Set Architecture (ISA) encoding to generate instruction input sequences that would stress-test different parts of the processor pipeline. We expect this enhancement to result in faster coverage than our current implementation.

VII. Acknowledgments

This material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7856. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) or the U.S. Government. This work is also partially supported by BU Hariri Research Incubation Award (#2020-06-005).

References

[1] "The Sodor processor: educational microarchitectures for RISC-V ISA," https://github.com/ucb-bar/riscv-sodor.
[2] "Verilator," https://github.com/verilator.
[3] A. Ahmed, F. Farahmandi, and P. Mishra, "Directed test generation using concolic testing on rtl models," in *DATE*, 2018, pp. 1538–1543.
[4] A. Ahmed and P. Mishra, "Quebs: Qualifying event based search in concolic testing for validation of rtl models," in *ICCD*, 2017, pp. 185–192.
[5] M. Aizatsky *et al.*, "Announcing oss-fuzz: Continuous fuzzing for open source software," *Google Testing Blog*, 2016.
[6] M. Böhme *et al.*, "Directed greybox fuzzing," in *CCS*, 2017.
[7] Cadence, "JasperGold Formal Verification Platform," 2019.
[8] M. Chen *et al.*, *System-level validation: high-level modeling and directed test generation techniques*. Springer Science & Business Media, 2012.
[9] M. Chen and P. Mishra, "Property learning techniques for efficient generation of directed tests," *IEEE TC*, vol. 60, no. 6, pp. 852–864, 2011.
[10] G. Dessouky *et al.*, "Hardfails: Insights into software-exploitable hardware bugs," in *USENIX Security*, 2019, pp. 213–230.
[11] Ekiwi, "ekiwi/rfuzz," https://github.com/ekiwi/rfuzz.
[12] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using bayesian networks," in *DAC*, 2003, pp. 286–291.
[13] C. Ioannides and K. I. Eder, "Coverage-directed test generation automated by machine learning–a review," *TODAES*, vol. 17, no. 1, pp. 1–21, 2012.
[14] A. Izraelevitz *et al.*, "Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations," in *ICCAD*, 2017.
[15] R. Kannavara *et al.*, "Challenges and opportunities with concolic testing," in *NAECON*, 2015, pp. 374–378.
[16] K. Laeufer *et al.*, "Rfuzz: coverage-directed fuzz testing of rtl on fpgas," in *ICCAD*, 2018, pp. 1–8.
[17] Y. Lyu *et al.*, "Automated activation of multiple targets in rtl models using concolic testing," in *DATE*, 2019, pp. 354–359.
[18] Microsoft, "microsoft/onefuzz," https://github.com/microsoft/onefuzz.
[19] R. Mukherjee *et al.*, "Hardware verification using software analyzers," in *IEEE Computer Society Annual Symposium on VLSI*, 2015, pp. 7–12.
[20] Sifive, "sifive-blocks," https://github.com/sifive/sifive-blocks/tree/master/src/main/scala/devices, 2020.
[21] G. Squillero, "Microgpan evolutionary assembly program generator," *GP/EM*, vol. 6, pp. 247–263, 2005.
[22] N. Stephens *et al.*, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, 2016, pp. 1–16.
[23] Synopsys, "Synthesis Design Compiler," https://www.synopsys.com/support/training/rtl-synthesis.html.
[24] Ucb-Art, "ucb-art/fft," https://github.com/ucb-art/fft.
[25] I. Wagner *et al.*, "Stresstest: an automatic approach to test generation via activity monitors," in *DAC*, 2005, pp. 783–788.
[26] L. Zhao *et al.*, "Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing." in *NDSS*, 2019.