RFUZZ: Coverage-Directed Fuzz Testing of RTL on FPGAs

Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach and Koushik Sen
Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, USA
{laeufer,jack.koenig3,dgkim,jrb,ksen}@eecs.berkeley.edu

ABSTRACT

Dynamic verification is widely used to increase confidence in the correctness of RTL circuits during the pre-silicon design phase. Despite numerous attempts over the last decades to automate the stimuli generation based on coverage feedback, *Coverage Directed Test Generation* (CDG) has not found the widespread adoption that one would expect. Based on new ideas from the software testing community around coverage-guided mutational fuzz testing, we propose a new approach to the CDG problem which requires minimal setup and takes advantage of FPGA-accelerated simulation for rapid testing. We provide test input and coverage definitions that allow fuzz testing to be applied to RTL circuit verification. In addition we propose and implement a series of transformation passes that make it feasible to reset arbitrary RTL designs quickly, a requirement for deterministic test execution. Alongside this paper we provide RFUZZ, a fully featured implementation of our testing methodology which we make available as open-source software to the research community. An empirical evaluation of RFUZZ shows promising results on archiving coverage for a wide range of different RTL designs ranging from communication IPs to an industry scale 64-bit CPU.

ACM Reference Format:
Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach and Koushik Sen. 2018. RFUZZ: Coverage-Directed Fuzz Testing of RTL on FPGAs. In *IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN (ICCAD '18), November 5–8, 2018, San Diego, CA, USA*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3240765.3240842

1 INTRODUCTION

While formal verification has shown promising results for exhaustively testing high quality designs [7], dynamic verification is still the most easily accessible and thus heavily used verification approach. Software simulators for RTL circuit designs are widely available and thus writing a test bench to simulate a new circuit design under various inputs is an easy way to gain confidence in its correctness. During or after the simulation, bugs are uncovered by manual waveform inspection, comparing the execution to a golden model or by various custom invariant (assertion) checkers. In order to ensure that a sufficient portion of the device under test (DUT) has been covered, various manual and automatic coverage metrics are used.

Once we rely on these end-to-end coverage metrics to measure the quality of test inputs, a natural research question that comes up is whether we can automate the stimuli generation with the goal of maximizing the coverage numbers with minimal manual developer intervention. When the coverage feedback is used to drive the input generation, this problem is known as *Coverage Directed Test Generation* (CDG). Various solutions have been proposed over the last two decades, however, we argue that they are either designed for a very narrow class of DUTs or require a good amount of expert time, such as for constructing a DUT-specific Bayesian network [3]. This might explain why generator-based approaches, which require the test engineer to manually specify biases from coverage reports, are still the most widely used technique today.

Over the last couple of years, coverage-guided mutational fuzz testing has emerged as one of the most effective testing techniques for finding correctness bugs and security vulnerabilities in real-world software systems [17]. This technique relies on the fact that many programs that are interesting to test can be run in a short amount of time with arbitrary bytes as input. The program under test is augmented with lightweight instrumentation that provides feedback on the coverage achieved by a particular input. Starting from one or several seed inputs, the fuzz engine tries to achieve new coverage by mutating previously discovered inputs. Once a new interesting input is discovered after running the program under test on it, it is added to the input pool to serve as a new starting point in the input space exploration. Compared to more formal techniques such as symbolic execution, fuzz testing has been able to scale up to much bigger real-world programs with smaller setup and engineering effort.

We believe that coverage-guided mutational fuzz testing is a new and interesting design point in the space of solution to the CDG problem. The approach of treating the test input as a series of bits or bytes allows this technique to be applied to a wide range of different circuits. Using FPGA-accelerated RTL simulation and synthesizable coverage feedback we can archive a high test execution speed similar to how some clever engineering techniques enabled fast fuzz testing speeds for software. In this regard, fuzz testing is—to the best of our knowledge—the first CDG technique to be designed specifically with FPGA emulation in mind [9].

In this paper we lay the ground work for applying coverage-guided mutational fuzz testing to the CDG problem: We define the test stimuli in such a way that mutation algorithms from software testing can be directly applied. We solve the problem of deterministic test execution in FPGA-accelerated simulation by introducing transfomations for `MetaReset` transformation and Sparse Memories. We define the notion of *Mux Toggle Coverage* that can be acquired during FPGA-accelerated simulation and used as feedback to the fuzz testing process. We empirically evaluate the performance of the proposed solutions on a variety of real world RTL circuits ranging from communication peripheral IPs to CPU cores. Finally we make our high-performance implementation of the proposed testing approach available to the research community as open-source software that can easily be used on a public cloud infrastructure for FPGA-accelerated fuzz testing experiments.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICCAD '18, November 5–8, 2018, San Diego, CA, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5950-4/18/11…$15.00
https://doi.org/10.1145/3240765.3240842

2 BACKGROUND

2.1 Dynamic Verification

In dynamic verification the *Device Under Test* (DUT) is executed within a *test bench* that instantiates and drives the DUT with concrete test inputs (*stimuli*). The RTL description of the DUT is compiled into a fast software simulation which is then executed on a general purpose CPU. The test bench can be written in a traditional *hardware description language* (HDL) like VHDL or Verilog, a special verification language like *e* or System Verilog, or in a traditional programming language like C. It is then linked to the simulator and executed in conjunction with the DUT.

Since dynamic verification looks at concrete executions of the design, a feedback mechanism is needed to know whether the simulated executions explore all interesting behaviors of the DUT. To this end, various notions of functional coverage have been defined [12]. Concrete functional coverage points or groups that are derived from the DUT specification need to be defined by the verification engineer. In addition to the user-defined functional coverage, code coverage metrics which are derived from the HDL implementation of the DUT are employed in the verification process. While high code coverage is not sufficient to declare the verification process a success, it can serve as an indicator of the verification progress and has been used in the past to evaluate automated stimuli generation approaches [5].

When using a software simulator for dynamic verification, the number of tests that can be executed daily is largely limited by the speed at which the design can be simulated. Especially for larger designs, FGPA based emulation or special purpose emulation hardware can significantly speed up the verification effort. Commercial special purpose accelerators support a wide range of test bench designs and coverage feedback. However, trying to support all techniques from software simulation on an accelerated platform makes them complicated and expensive. FPGA-based emulation on the other hand is more easily available with FPGAs now available for rent on public cloud infrastructure and open-source tool support [8]. However, test bench design becomes more difficult in this case since non-synthesizable constructs that are often used in stimuli generation and scoreboards cannot easily be mapped to FPGAs. Thus traditional test bench designs cannot easily be used and some design trade-offs need to be reevaluated.

2.2 Coverage-Directed Mutational Fuzz Testing

In this section we introduce the basic *coverage-directed mutational fuzz testing* components and algorithm as used by the popular software fuzzer AFL [17] and various work that builds on top of it. A coverage-directed mutational fuzz testing tool (fuzzer) consists of three components: (1) A fuzz server that snapshots the program under test and quickly resets it before every test. (2) A static or dynamic instrumentation pass that augments the program under test to provide feedback about its behavior during execution. (3) A fuzz engine which implements the algorithms to select parent inputs, mutate them, and analyze the feedback from the instrumentation.

The program under test (PUT) is modeled as a pure function that takes an arbitrary length byte array as input. The behavior of the PUT should only depend on the selected input and thus a given input precisely describes a test execution. In order to guarantee that

Algorithm 1 Coverage-guided mutational fuzzing

Given: program p, set of initial inputs I
Returns: a set of generated test inputs

1: $\mathcal{S} \leftarrow I$
2: $totalCoverage \leftarrow \emptyset$
3: **repeat**
4:   **for** *input* in $\mathcal{S}$ **do**
5:     **for** $1 \leq i \leq$ NUMCANDIDATES(*input*) **do**
6:       $candidate \leftarrow$ MUTATE($input, \mathcal{S}$)
7:       $coverage \leftarrow$ RUN($p, candidate$)
8:       **if** $coverage \not\subseteq totalCoverage$ **then**
9:         $\mathcal{S} \leftarrow \mathcal{S} \cup \{candidate\}$
10:         $totalCoverage \leftarrow totalCoverage \cup coverage$
11: **until** given time budget expires
12: **return** $\mathcal{S}$

all test results are reproducible, it is imperative that all program state is reset in between tests. The fuzz server uses memory snapshot techniques to perform the reset before rerunning the PUT with a new input provided by the fuzz engine.

The goal of the instrumentation pass is to augment the program in such a way as to provide coverage feedback for every execution. The feedback is used by the fuzz engine to guide its search of the input space and to come up with a test corpus that achieves a high coverage. In order to be effective, the chosen coverage metric needs to be lightweight enough to not slow down test execution significantly, detailed enough to be able to guide the fuzz engine, but also abstract enough to not overburden the fuzzer with every little detail of the PUT behavior. The coverage feedback used by AFL that has shown to be successful in practice for software testing is an approximate version of branch coverage: During instrumentation every basic block in the PUT is assigned a random ID. Once the program takes a transition in the control flow graph, the source and destination ID are hashed together and used to index into a 65536 entry table of 8-bit counters. The selected counter entry is then incremented. In a post processing step, the counter values which range from 0 to 255 are placed into 8 exponentially increasing

Name	Description
bitflip 1/1	flip single bit
bitflip 2/1	flip two adjacent bits
bitflip 4/1	flip four adjacent bits
bitflip 8/8	flip single byte
bitflip 16/8	flip two adjacent bytes
bitflip 32/8	flip four adjacent bytes
arith 8/8	treat single byte as 8-bit integer, add/sub values from 0 to 35
arith 16/8	treat two adjacent bytes as 16-bit big/little endian integer, add/sub values from 0 to 35
arith 32/8	treat four adjacent bytes as 32-bit big/little endian integer, add/sub values from 0 to 35

Figure 1: Deterministic Mutation Techniques

buckets. The intuition behind this is that traversing an edge twice instead of once is new and interesting, whereas going from 6 to 7 transitions is normally not relevant. This technique has good accuracy for small to medium sized programs, is relatively easy to implement, and has an acceptable performance overhead.

At the core of a fuzz testing system, the fuzz engine is responsible for selecting new test inputs to be evaluated and analyzing the resulting coverage feedback from the PUT. The algorithm (shown in Algorithm 1) starts with an initial set of *seed* inputs, e.g., a set of small PNG images when testing a PNG parser. Sometimes a single empty input is used as a starting point. All seed inputs are placed in the test set data structure S. In the main loop, the fuzz engine selects one input from S and applies a set of mutations to it. Each result of a mutation is executed by the fuzz server and the resulting coverage is analyzed. If a new coverage point is reached, the input that caused it, is added to S. After a user controlled timeout, the fuzzing process terminates and the inputs in S can be used as a test corpus.

The mutation algorithm uses two kinds of mutators: deterministic and non-deterministic. A mutator is a function that takes a test as input and modifies it in order to generate several new *child* tests. An example for a deterministic mutator in AFL is the `bitflip 1/1` mutation which generates one child per bit in the parent input with the corresponding bit inverted. A list of all deterministic mutators that are relevant to this paper can be found in Figure 1. The non-deterministic mutations are performed in the so called `havoc` stage of AFL. In each application of the `havoc` mutation, between 2 and 128 random mutations are performed on the parent input. A list of all possible submutations is presented in Figure 3. While deterministic mutations mutate every position in the input, non-determinist mutators randomly chose the position to mutate.

The main advantages of coverage-directed mutational fuzzing compared to more formal techniques such as symbolic execution are the smaller engineering effort, better scalability to large real world programs such as web browsers, and the portability of the approach due to its relative simplicity. The generated test corpus contains inputs that tend to be small. If a bug is uncovered while fuzzing (e.g., if a program crash is observed), the test input serves as a witness of the bug and can be used to debug the problem. The simple input definition of an array of bytes works on a wide range of programs and the coverage feedback is carefully engineered for good test performance.

3 FUZZ TESTING OF RTL CIRCUITS

While coverage-directed mutational fuzz testing is an input generation technique that uses coverage feedback, it cannot be directly

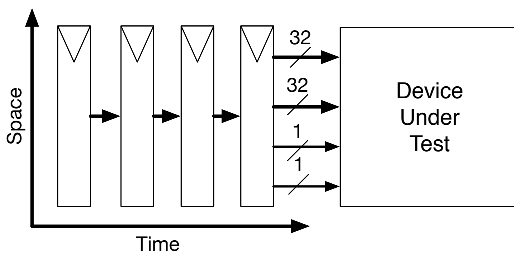


Figure 2: Input Definition

Name	Description
bitflip	flip a random bit
interest 8	overwrite a random 8-bit integer with interesting value
interest 16	overwrite a random 16-bit integer with interesting value
interest 32	overwrite a random 32-bit integer with interesting value
arith 8/8	treat random single byte as 8-bit integer, add/sub one value from 0 to 35
arith 16/8	treat two random adjacent bytes as 16-bit big/little endian integer, add/sub one value from 0 to 35
arith 32/8	treat four random adjacent bytes as 32-bit big/little endian integer, add/sub one value from 0 to 35
random 8	overwrite random byte with random value
delete	delete a random sequence of bytes
clone	clone a random sequence of bytes
overwrite	overwrite a random sequence of bytes

Figure 3: Non-Deterministic havoc Mutations

applied to the CDG problem for hardware designs: A digital circuit is not a binary file format parser that can read an arbitrary number of bytes. Instead, it has a number of input wires that can take different values in each cycle. In addition to that, the memory snapshotting techniques used to reset software to a known state before each test [17] cannot be directly applied to FPGA-accelerated RTL simulation. Instead we need a way to quickly reset RTL state without changing the behavior of the DUT. Furthermore, while its notion exists in HDLs for RTL simulation, branch coverage does not directly apply to RTL designs. Branches in the HDL source code are mapped to multiplexers in the circuit which output one of two input values during each cycle, which is different from sequential software where only one branch is active at a given point in time. In this section we therefore discuss how a test input for RTL circuits can be defined, the work necessary to make DUTs resettable on the FPGA, and how the notion of branch coverage can be translated to testing RTL circuits.

3.1 Input Definition

RTL circuits are commonly represented as module hierarchy featuring one top-level module that will be connected to the test harness or external pins. In our methodology, the top-level input pins are connected to the testing tool. We concatenate all input pins and map the resulting bit vector to a series of bytes representing the input values in one particular test cycle. In order to allow the fuzz engine to apply a different value during each test cycle, we simple concatenate single cycle test inputs to form a multi-cycle input. We thus concatenate inputs in space and time as illustrated in Figure 2. The number of cycles of a particular test runs for is thus determined by the number of *test input* bits divided by the number of input bits to the top-level module of the DUT. Since the DUT is reset to a known state before each test execution, the test inputs fully describe the test execution. Reproducing coverage or assertion violations thus only requires knowledge of the DUT and the test input.

3.2 Deterministic Test Execution

In order to make tests deterministic and repeatable, fuzz testing requires the program or device under test to be started from a known state. This ensures that only the test inputs affect the behavior that will be observed by the fuzz engine and thus a test can be fully reproduced as long as the inputs are known. Resetting a program to a known state can require a non-trivial amount of time. The simplest approach, to restart the program under test for every test invocation, includes the cost of loading the program into memory and process creation which limits the number of test executions per second. The popular fuzzer AFL takes advantage of the `fork` system call and copy-on-write optimization by the operating system to reduce the overhead.

Quickly resetting an RTL circuit mapped onto an FPGA poses its own set of challenges which need to be addressed in order to apply fuzz testing to this domain. In the following sections we discuss two major problems and how they are solved in our work: (1) for efficiency reason, many registers are not reinitialized during device reset; (2) memories do not feature reset circuitry and can only be initialized one word at a time

3.3 Register Meta Reset

Reset circuitry for registers takes up space on the wafer and is thus omitted from designs whenever possible. The initial value of these registers is thus undefined when the DUT comes out of reset. Classic circuit simulators deal with this fact by introducing an X value which marks an uninitialized wire (in 4-state simulation) or by randomizing the initial values of the register (in 2-state simulation). Since this work is targeted at FPGA-accelerated simulation, we would like to make use of a 2-state solution. Randomizing the register values on the FPGA however is also a non-trivial endeavor and could lead to sporadic tests. Instead, there are two promising solutions: (1) we can treat the initial register state as part of the input and load the values through a scan chain before each test (2) we can reset all registers to a predefined value before each test.

In this work we implement solution 2 in a transformation pass that works on the intermediate representation (IR) of the circuit and adds a `MetaReset` wire which resets all registers in the circuit to zero. As an example Figure 4 illustrates the addition of a `MetaReset` to a register description in Verilog. Our test harness thus applies the following sequence before each individual test: First, the `MetaReset` is activated for one cycle in order to initialize each register to zero. Next the `MetaReset` is released and the actual `Reset` of the DUT is asserted in order to take the device through the reset procedure envisioned by its designer. During this phase it is essential to provide deterministic inputs to the DUT because a register might be hardwired directly to a top-level input in the original design. We again chose all zeros as a deterministic input. This approach is sound since starting the design with all registers set to zero is allowed by the RTL (an X can be any value), but incomplete since other possible values are not explored.

3.4 Sparse Memories

Memories are rarely meant to be reset when the circuit is turned on. The concept of a memory can be mapped to a much more area efficient implementation compared to a register, because only a

```
reg [31:0] r;
always @(posedge clk) begin
  if (reset) begin
    r <= 32'h1993;
  end else begin
    r <= r_next;
  end
end
```

(a) Register With Reset

```
reg [31:0] r;
always @(posedge clk) begin
  if (metaReset) begin
    r <= 32'h0;
  end else begin
    if (reset) begin
      r <= 32'h1993;
    end else begin
      r <= r_next;
    end
  end
end
```

(b) Register With MetaReset

Figure 4: Meta Reset Transformation

small number of memory words (bounded by the number of write ports) can be updated in each cycle. This restriction sets the lower bound for the number of cycles needed to fully reset a memory to be the memory size in words divided by the number of write ports. This number can be very high in practice, especially when considering the data or program memory of a processor design.

The task of resetting memories is not as difficult as the upper bound mentioned above might suggest. To see why it is important to consider the details of the fuzz testing scenario proposed in this paper: The core idea is to mutate the seed inputs as often as possible and to evaluate every generated input on the instrumented DUT. In order for this to be feasible, the test size and thus the number of cycles it takes to execute a given test is relatively small. Thus the number of *writes* that may occur during a single test for a given memory is bound by the number of write ports times the number of test cycles. If we can keep track of the memory locations that have been written in a test execution, we are able to undo all the changes made and thus reset the DUT on the FPGA. This solution would require a memory for every write port to remember the addresses that have been written to in addition to the actual memory that stores the values. In the worst case, resetting this kind of memory would take as many cycles as the previous test took to execute.

Going back to the observation that we will only observe a small number of writes, we can refine the design of our resettable memory: Since the number of writes is small, most memory locations will contain the reset value which we define to be zero, just as for registers. The read port of such a *sparse memory* needs to work in the following manner: If the address that is requested has been written to, return the last written value. If the address that is requested has never been written to, it is uninitialized and thus we need to return zero. We can keep track of the addresses that have been written to by using a content addressable memory (CAM). The CAM can also be used to map the requested address to a much smaller memory that only needs to be able to hold as many values as may be written during a maximum length test. Thus, this kind of *sparse memory* needs often times much less SRAM than the original version. The CAM can easily be reset in a single cycle by connecting the valid bit of each entry to the `Reset` signal of the DUT. Implementing a custom transformation pass on the RTL of the DUT, we can automatically replace memories by a sufficiently large *sparse memory*.

Using the *MetaReset* and the *SparseMem* transformations we can ensure that the DUT can be reset to a fully deterministic state in only two clock cycles. This allows for rapid and repeatable test execution, thus enabling fuzz testing of RTL circuits on FPGAs.

3.5 Coverage Definition

We require precise definitions of our coverage metrics for two reasons: (1) To define an end-to-end metric that can be used to measure how well our implementation of mutational fuzz testing for RTL compared to a baseline technique; (2) To define an intermediate coverage metric that serves as feedback to the fuzz engine in order to guide the search of the input space. While prior industrial work [2, 14, 15] uses functional coverage models manually specified by verification engineers, these test suites are expensive to create and thus generally unavailable to the broader research community. Instead, we focus on automatic coverage that can be derived directly from our suite of open-source benchmark circuits. This kind of coverage has been used in related academic [13] and industrial [5] work.

Most automatic coverage definitions focus on the description of the circuit expressed in a common HDL like Verilog or VHDL [12]. However, our system works with any RTL circuit regardless of the hardware description or generation language it is written in. We thus define our coverage metric in relation to the synthesizable structure of the circuit as represented in an HDL-agnostic IR. This also ensures that we can synthesize and thus collect coverage information during FPGA-accelerated simulation.

We look at *mux control coverage* which treats each 2:1 multiplexer select signal as an independent cover point. Multiplexers with more than two inputs can be trivially converted into a series of 2:1 multiplexers. We chose this metric as it can be automatically applied to any RTL circuit as long as the multiplexers are explicitly modelled. It is also well suitable for FPGA-accelerated simulation since we do not need any additional circuitry to evaluate the cover points.

For a mux control condition to be fully covered, we require that it evaluates to true as well as to false during a single test. On the FPGA, this requires only minimal additional hardware — two 1 bit registers and two 1 bit multiplexers — to remember the observed values. We combine the coverage observed in multiple tests by calculating the union of mux control conditions covered. Note that by this definition it is not enough for a condition to always be true in one test and always be false in another test to be counted as covered. Instead, both values need to be observed in a single test.

The *coverage* used in Algorithm 1 is thus defined to be the set of multiplexers in the DUT which had their control signal toggle during test execution. If a test input manages to toggle a mux control signal that had never been toggled before, it is considered interesting and is added to the test data structure S.

3.6 Mutation Algorithms

Our input definition allows us to directly implement the mutation heuristics from the successful AFL fuzz testing tool [17] presented in Section 2.2. Similar to AFL, every new entry in our test set is first mutated with the deterministic mutation techniques listed in Figure 1. Once we run out of deterministic mutations to apply, we switch to our implementation of the AFL `havoc` stage which makes use of the mutations presented in Figure 3. For any mutation that changes the size of the input array, we pad with zeros when necessary in order to maintain an input size that is a multiple of the bytes needed in a single cycle.

3.7 Constrained Interfaces

In hardware testing, many interfaces make assumptions that have to be respected by the stimuli generator. An example for this is a memory bus which can rely on the fact that any participant will respect the protocol specification. To this end, a test input generator in traditional directed random testing needs to be implemented in such a way as to not violate the guarding assumptions. A similar solution could be applied to fuzz testing by implementing an RTL adapter that takes the unconstrained inputs from the fuzzer and — by construction — generates valid bus transactions from them.

However, the feedback-directed manner of the fuzzing approach allows for a more convenient solution: We observe that modern HDLs allow designers to specify interface constraints through `assume` statements in the DUT source code. In our benchmarks which make extensive use of the TileLink bus, this mechanism is used to implement a synthesizable bus monitor which detects invalid transactions. Taking the conjunction of all assumptions in the monitor over all cycles in a test, we can derive a binary signal which indicates whether the given test inputs exercise the DUT in a valid manner. This *valid* signal is included by the test harness on the FPGA with the regular mux condition coverage as feedback to the fuzz engine. The simplest way of using this signal is to reject all invalid inputs before updating the coverage map. This is comparable to rejection sampling in random directed testing.

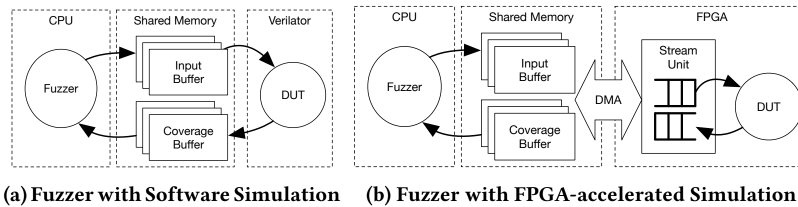
The authors of the open source Java fuzz testing tool JQF [1] have extended the core fuzz testing method described in Algorithm 1 to take advantage of the feedback regarding the validity of a generated test input. They keep two separate coverage maps: one for the total coverage and one for the coverage achieved by valid inputs only. A new input is added to the test set S when it achieves new total coverage or if it is valid and achieves new valid coverage. This extension allows the fuzz engine to discover valid inputs from invalid ones. We implemented the JQF technique using two coverage maps in our testing tool.

4 IMPLEMENTATION

We implemented the proposed testing methodology in an open-source tool called RFUZZ[2]. The first part of our tool is an instrumentation and harness generation component, which works on arbitrary RTL circuits described in the FIRRTL IR [6]. It automatically generates a test harness for software or FPGA-accelerated simulation. The second part of our tool is the actual input generator which connects to the test harness running in software or on the FPGA to provide DUT inputs and analyse the resulting coverage.

Our tool is language-agnostic since it can work on arbitrary RTL designs expressed in the FIRRTL IR [6]. Once a target design is translated into FIRRTL IR from its source HDL, we can apply compiler passes for the target RTL regardless of its source HDL. RFUZZ is also fully automated as the target RTL is instrumented through compiler passes and the fuzzer uses the target information generated by the compiler. Only some parameters to the fuzzer such as the mutation technique and seed inputs to use need to be specified by the user.

[1] https://github.com/rohanpadhye/jqf
[2] https://adept.eecs.berkeley.edu/papers/rfuzz



(a) Fuzzer with Software Simulation **(b) Fuzzer with FPGA-accelerated Simulation**

Figure 5: Share Memory Implementations for Communication between the Fuzzer and the Test Harness

4.1 Instrumentation

Custom transforms are implemented as compiler passes that plug into the FIRRTL compiler. We use the compiler's dead code elimination and constant folding to minimize the redundant expressions before instrumenting the coverage signals. This helps us keep the size of the automated coverage feedback as small as possible.

The *MuxCov* (Section 3.5) pass automatically identifies intermediate coverage wires by traversing the circuit description. Coverage wires that are automatically identified by the circuit traversal are then rewired through the module hierarchy to be available as outputs of the top-level module so that coverage wire values are observed by the test harness. This pass also generates a meta data file that contains the information about the coverage and the input pins in the RTL design for the test harness generation.

The details of our *MetaReset* and *SparseMem* passes are explained in Section 3.3 and Section 3.4, respectively.

4.2 Test Harness Generation

The test harness generator automatically generates a wrapper for any RTL design by consuming the target design information, including input and coverage pins generated, by the instrumentation passes.

It instantiates the instrumented DUT and connects the coverage pins inserted by the instrumentation pass to toggle detection circuitry. It also automatically derives a buffer format definition for the required input and coverage size and emits Verilog as well as C++ code for the software and FPGA-accelerated simulation environments to interface with the buffers.

The test harness is further automatically transformed by FIRRTL compiler passes for efficient token-based simulation on the FPGA [8], dramatically reducing manual effort for FPGA-accelerated simulation. Our tool also automatically generates the buffer stream unit mapped on the FPGA and integrates it with the test harness for communication with the fuzzer.

Finally, the test harness generator emits the target-specific information about coverage counters, top-level inputs, and buffer formats, which is consumed by the fuzzer to test a particular circuit design.

Name	Input Width	Mux Cover Points	Lines of FIRRTL
Sodor1Stage	35	714	3617
Sodor3Stage	35	746	4021
Sodor5Stage	35	945	4088
I2C	165	301	2373
SPI	167	323	4046
FFT	259	195	1545
Rocket Chip	239	4517	43856

Table 1: Benchmarks

4.3 Fuzzer

Whereas the DUT and the coverage counters can be synthesized onto an FPGA, the input generation and coverage analysis is performed by a fast fuzzer on the CPU. Implementing this part in software allows for greater flexibility to investigate new mutation and feedback strategies. While an integrated solution on the FPGA could be even fast, we achieve good performance with a high bandwidth DMA channel.

Figure 5 shows how the fuzzer efficiently communicates with the test harness through share memory buffers. Note that the fuzzer is unaware of whether the test harness is run in software simulation or on the FPGA. The fuzzer allocates multiple buffers in the shared memory region, and test inputs and coverage feedback are batched to the buffers (Figure 5(a)). When the test harness is run in software simulation, the software simulator directly accesses these buffers. To cope with high round-trip latency between CPU and FPGA, when the test harness is run in the FPGA, theses buffers are transferred through a high bandwidth DMA to the buffer stream unit that post-processes the data in the buffers (Figure 5(b)).

5 EVALUATION

We evaluate the proposed testing methodology using our RFUZZ tool on a range of open-source RTL designs:

1. **TileLink Peripheral IP**: These consist of a SPI and I2C peripheral IP which are used in the commercial SiFive Freedom SoC platform [3]. They interface with the fuzzer through a TileLink port which includes a synthesizable bus monitor. The feedback from the monitor is used to ensure that only valid inputs are included in the reported coverage.
2. **FFT**: As an example of a DSP block, we use a FFT implementation produced by an open-source FFT generator [4].
3. **RISC-V Sodor Cores**: We selected three different educational RISC-V cores maintained by the LibreCores project [5]. In order to directly affect the executed instructions, we create a special fuzz testing top-level module which — instead of instantiating a scratchpad memory — directly wires the instruction memory interface to the top-level inputs. This allows our testing tool to act as the instruction memory and directly supply the core with instructions to execute.
4. **RISC-V Rocket Core**: In order to test the scalability of our approach, we use the RISC-V Rocket Chip [1] as our final benchmark. This 64-bit in order core is supported by industry and is able to boot the Linux operating system. Its size impacts the execution speed of software simulation and thus

[3] https://github.com/sifive/sifive-blocks
[4] https://github.com/ucb-art/fft
[5] https://github.com/ucb-bar/riscv-sodor

	Sodor3Stage	Rocket
Verilator	345 kHz	6.89 kHz
FPGA	1.7 MHz	1.46 MHz
Speedup	4.9x	212x

Table 2: Speedup: FPGA vs Software Simulation

allows us to evaluate the benefits of an FPGA-accelerated simulation approach to the CDG problem.

A detailed list of the benchmarks is available in Table 1.

For our evaluation we use software simulation on the public AWS cloud infrastructure to quickly evaluate various configurations in parallel. Each fuzz testing run was performed on its own virtual core. Since several of the proposed mutation techniques make random decisions when generating new test inputs, we rerun experiments four times with different seeds to the pseudo random number generator and average the results.

During each testing run, we save all generated inputs that make it into the test set S to disk. In order to evaluate the achieved coverage independently from our testing tool, we use a series of Python scripts to calculate end-to-end coverage metrics. Since speed does not matter in this context, we are able to restart the software simulation for each entry can thus be confident that various tests are indeed independent and do not affect each other. The end-to-end analysis scripts also exclude any invalid inputs as indicated by an assumption failure during the test run. We can thus ensure that — independent from RFUZZ — our coverage numbers only include valid inputs as checked by the monitors and assume statements. All coverage in this section is measured as a fraction of the maximum mux control toggle coverage as indicated by the number of multiplexers in the design. It might be impossible for some mux control wires to be influenced from the inputs controlled by the fuzzer and thus there is no guarantee that 100 % coverage can be achieved.

5.1 Comparison to Random Testing

Similar to our proposed technique, random testing is applicable to any RTL circuit without DUT specific setup costs. We implement random testing in our tool in order to measure whether coverage-directed mutational fuzz testing provides any advantages over the simple random baseline. In order to implement the baseline we replace the normal mutation algorithms that modify a given test input to instead generate a new independent random input. Figure 6 shows the results for all of our benchmarks. As we can see, the random baseline quickly saturates, whereas the coverage-guided fuzz testing is able to make progress by mutating previously discovered inputs.

5.2 Constraint Interfaces

As explained in Section 3.7 we can deal with constrained interfaces by observing the assumption failures that result from invalid test inputs. All tests in Figure 6 used the JQF technique to generate valid inputs. As we can see, this provides a significant improvement over the random baseline for the TileLink I2C and SPI peripherals.

	Local Machine (Verilator)	Amazon F1 (FPGA)
CPU	AMD Ryzen 7 1700X	8 vCPUs
Memory	32 GB	122 GB
FPGA	-	Xilinx UltraScale+ VU9P
DMA bandwidth	-	1.5 GB/s

Table 3: Machine Specifications for Speedup Evaluations

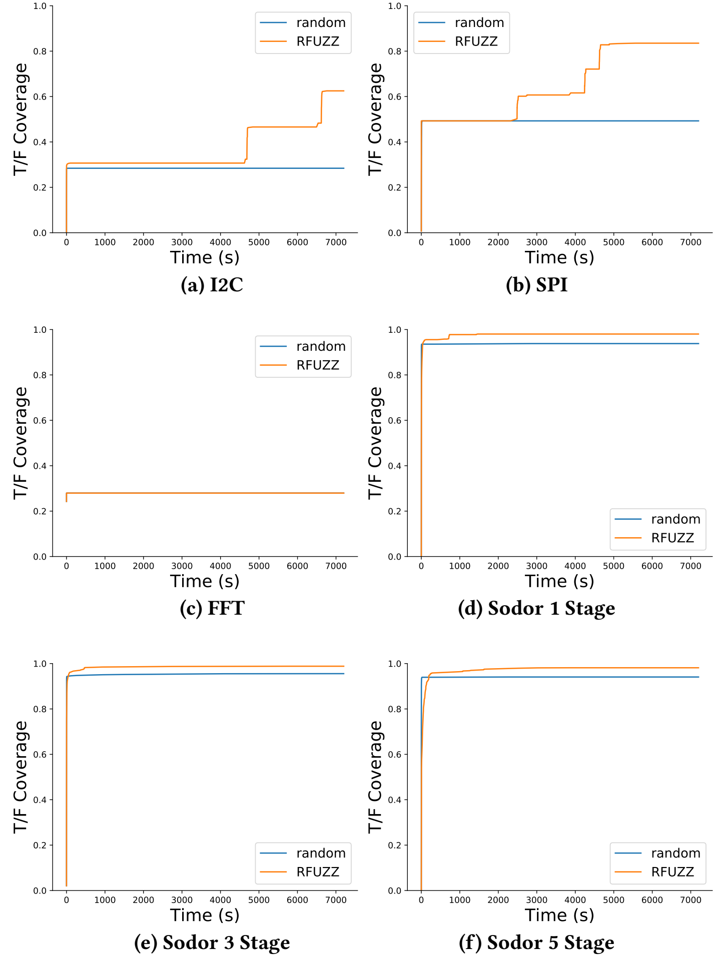


Figure 6: Mux Control Toggle Coverage over Time: RFUZZ vs. Random Testing

5.3 Software vs. FPGA-Accelerated Simulation

While for small circuits, generating the test inputs and analyzing the resulting coverage takes the majority of time, this changes as the design becomes bigger. For large designs such as a real-world 64-bit processor, the simulation time is the major bottleneck. As mentioned throughout Section 3, we took specific care to design our testing methodology in such a way that the device under test can be simulated on the FPGA.

To show how FPGA-accelerated simulation enables us to scale to testing complete large-scale systems we measured the execution speed for a small educational processor (Sodor3Stage) and a productized in-order processor (Rocket). Table 3 shows the specifications for the machines we used in this evaluation. We compiled bitstreams for FPGA-accelerated simulations using Vivado 2017.1 and both designs closed timing at 75 MHz. The FPGA synthesis time was 2~5 hours.

Table 2 shows the speedup of FPGA-accelerated simulation over software simulation for two designs. As expected, we can achieve significant speedup for a complex design, but even a small design can benefit from FPGA-accelerated simulation. Notably, software simulation slows down significantly with complex designs while

FPGA-accelerated simulation provides high simulation rates regardless of design complexities. With FPGA-accelerated simulation, the simulation speed is bottlenecked by the speed at which our fuzzing software analyses coverage and generates new inputs. Thus shifting some of that functionality from the fuzzer to the FPGA could significantly improve the simulation rates in the future.

6 RELATED WORK

The prior work most similar to RFUZZ is MicroGP [13] which focuses on maximizing statement coverage in the HDL description of various processor implementations. It uses an instruction template library in order to generate and recombine programs which allows for more powerful mutation techniques but increases the amount of setup work needed. It also restricts this line of work to the domain of processor testing, whereas coverage-directed fuzz testing also shows promising results when testing various communication IP. Another difference is that MicroGP is clearly targeted towards slow DUT execution in a software simulation. An industrial evaluation reports that simulation takes 30 times more resources than the core genetic algorithm [5]. RFUZZ on the other hand is geared towards fast FPGA-accelerated simulation, using a simpler algorithm to keep up with the test execution speed provided by such a platform.

There are various other approaches to the CDG problem that do not rely on a modified genetic algorithm. Tarsiran et al.[14] analyse the circuit in order to improve the biases for an existing random input generator. Our work on the other hand does not assume that a generator exists. Nativ et al. [10] propose a system that uses coverage feedback to direct a random input generator. However, this system also relies on rules unique to a single DUT that need to be specified by a verification expert. Fine et al. [3] present coverage directed test generation using Bayesian Networks to guide the input generation. While the network weights are learned automatically, the network topology is DUT specific and needs to be designed by a verification engineer. Wang et tal. [16] use a manually designed abstract model of the DUT to automatically generate inputs that maximize coverage.

`bluecheck` [11] is a synthesizable test bench framework that takes advantage of the BlueSpec HDL. Similar to our work, it is directly targeted at FPGA-based testing. However the authors report some issues trying to reproduce failing test cases discovered with the FPGA emulation. Our system employs the *MetaReset* and *SparseMem* techniques in order to ensure that the FPGA-accelerated simulation results are deterministic. While `bluecheck` depends on the BlueSpec HDL, RFUZZ is HDL-agnostic. Our work focuses only on maximising RTL coverage whereas `bluecheck` can also generate checks to find design bugs while running on the FPGA.

The technique presented by Gent et al. [4] is fully automated and also works directly on the RTL circuit just like RFUZZ but uses software simulation only, which may not scale to larger designs. The extended coverage metrics proposed by the authors could be added to our fuzzing setup in order to provide more detailed guidance to our fuzzer.

7 CONCLUSION

In this paper we show how coverage-directed mutational fuzz testing can be used to automatically test arbitrary RTL circuits on FPGAs. We provide a high-performance implementation of this technique. Our evaluation shows consistent improvements over random testing, especially for circuits that provide feedback on test input validity.

ACKNOWLEDGMENTS

We would like to thank Rohan Padhye and Caroline Lemieux for explaining to us how their JQF fuzzer deals with invalid inputs. Research partially funded by NSF grants CCF-1409872 and CCF-1423645; DARPA CRAFT HR0011-16-C-0052 and HR0011-12-2-0016; Intel Science and Technology Center for Agile Design; and ADEPT Lab industrial sponsors and affiliates Intel, Google, Siemens and SK Hynix. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and does not necessarily reflect the position or the policy of the sponsors.

REFERENCES

[1] Krste Asanović, Rimas Avižienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Palmer Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Benjamin Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator.* Technical Report UCB/EECS-2016-17.
[2] Mrinal Bose, Jongshin Shin, Elizabeth M Rudnick, Todd Dukes, and Magdy Abadir. 2001. A genetic approach to automatic bias generation for biased random instruction generation. In *Proceedings of the 2001 Congress on Evolutionary Computation.*
[3] Shai Fine and Avi Ziv. 2003. Coverage directed test generation for functional verification using bayesian networks. In *DAC '03.*
[4] Kelson Gent and Michael S Hsiao. 2016. Fast Multi-level Test Generation at the RTL. In *VLSI '16.*
[5] Charalambos Ioannides, Geoff Barrett, and Kerstin Eder. 2010. Feedback-based coverage directed test generation: An industrial evaluation. In *Haifa Verification Conference.* Springer, 112–128.
[6] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. 2017. Reusability is FIRRTL Ground: Hardware Construction Languages, Compiler Frameworks, and Transformations. In *ICCAD '17.*
[7] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, et al. 2009. Replacing Testing with Formal Verification in Intel (c) Core (TM) i7 Processor Execution Engine Validation. In *CAV '09.*
[8] Donggyu Kim, Adam Izraelevitz, Christopher Celio, Hokeun Kim, Brian Zimmer, Yunsup Lee, Jonathan Bachrach, and Krste Asanović. 2016. Strober: fast and accurate sample-based energy simulation for arbitrary RTL. In *ISCA '16.*
[9] A. Nahir, A. Ziv, and S. Panda. 2012. Optimizing test-generation to the execution platform. In *17th Asia and South Pacific Design Automation Conference.*
[10] Gilly Nativ, S Mittennaier, Shmuel Ur, and Avi Ziv. 2001. Cost evaluation of coverage directed test generation for the IBM mainframe. In *International Test Conference 2001.*
[11] Matthew Naylor and Simon Moore. 2015. A generic synthesisable test bench. In *MEMOCODE '15.*
[12] Andrew Piziali. 2007. *Functional verification coverage measurement and analysis.* Springer Science & Business Media.
[13] Giovanni Squillero. 2005. MicroGP - an evolutionary assembly program generator. *Genetic Programming and Evolvable Machines* 6, 3 (2005), 247–263.
[14] Serdar Tasiran, Farzan Fallah, David G Chinnery, Scott J Weber, and Kurt Keutzer. 2001. A functional validation technique: biased-random simulation guided by observability-based coverage. In *ICCD 2001.*
[15] Shmuel Ur and Yaov Yadin. 1999. Micro Architecture Coverage Directed Generation of Test Programs. In *DAC '99.*
[16] J. Wang, H. Li, T. Lv, T. Wang, X. Li, and S. Kundu. 2016. Abstraction-Guided Simulation Using Markov Analysis for Functional Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 2 (2016), 285–297.
[17] Michał Zalewski. 2014. American Fuzzy Lop Technical Details. http://lcamtuf.coredump.cx/afl/technical_details.txt. (2014). Accessed April, 2018.