CirFix: Automatically Repairing Defects in Hardware Design Code

Hammad Ahmad, Yu Huang, Westley Weimer
University of Michigan, Ann Arbor, MI
Bugs = Expensive

“Debugging, on average, has grown to consume more than 60% of today’s ASIC and SoC verification effort.”
-Harry Foster, Mentor Graphics Corporation

Intel does its best to tamp down impact of Spectre and Meltdown in earnings call

Google lost $1.7M in ad revenue during YouTube outage, expert says

Amazon’s one hour of downtime on Prime Day may have cost it up to $100 million in lost sales
A Solution in the Realm of Software: Automated Program Repair (APR)

Faulty program w/ bug(s) → Fault localization

Test suite w/ at least one failing test → Patch

Fault localization → Patch → Validation

Repair program

OR

No Repairs Found

“Generate and Validate”
Problem #1: Software-based APR is not amenable to traditional hardware testbenches

Test suite with two failing tests

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>tc0</td>
<td>pass</td>
</tr>
<tr>
<td>tc1</td>
<td>pass</td>
</tr>
<tr>
<td>tc2</td>
<td>fail</td>
</tr>
<tr>
<td>tc3</td>
<td>pass</td>
</tr>
<tr>
<td>tc4</td>
<td>fail</td>
</tr>
<tr>
<td>tc5</td>
<td>pass</td>
</tr>
<tr>
<td>tc6</td>
<td>pass</td>
</tr>
<tr>
<td>tc7</td>
<td>pass</td>
</tr>
</tbody>
</table>

Fitness = 0.75 (6 passing, 2 failing tests)

Compiler version N-2017.12-SP2-1_Full64; Runtime version N-2017.12-SP2-1_Full64; Jan 11 11:37 2021

<table>
<thead>
<tr>
<th>time, clk, reset, enable, count_out, overflow_out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, 0, 0, 0, x, x</td>
</tr>
<tr>
<td>5, 1, 0, 0, x, x</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>250, 0, 0, 1, 5, 1</td>
</tr>
<tr>
<td>255, 1, 0, 1, 5, 1</td>
</tr>
<tr>
<td>256, 1, 0, 1, 6, 1</td>
</tr>
</tbody>
</table>

$finish called from file "first_counter_tb_t3.v", line 70.
$finish at simulation time 258

Fitness = ???
Problem #2: Fault localization approaches from software-based APR do not scale to hardware

Faulty program
Passing tests
Failing tests

Fault localization techniques from software-based program repair (e.g., Tarantula)

Ranked suspiciousness ratings based on lines visited for passing and failing tests

But...

Hardware designs are \textit{parallel} in nature!
Introducing: *CirFix*

*CirFix*: A hardware-design focused automated repair algorithm based on genetic programming (GP)

- Proposes a novel dataflow-based fault localization approach for hardware designs to implicate faulty design code
- Presents a novel approach to guide the search for a hardware design repair using the existing hardware verification process
- *SPOILER*: Fixes hardware defects with a repair rate similar to that of established software-based APR techniques
Fitness Function

- Fitness scores to evaluate candidate repairs
- Testbench *instrumentation* to record the values of wires and registers at specified timesteps
- *Bit-level comparison* of instrumented wires and registers against expected behavior
Fitness Function: Comparison

**Oracle**: a developer-provided information for circuit behavior

\[ \text{sum}(S, O) = \sum_{t=0}^{k} \sum_{b=0}^{n} \begin{cases} 1 & (O_{t,b}, S_{t,b}) \in \{(0,0), (1,1)\} \\ \phi & (O_{t,b}, S_{t,b}) \in \{(x,x), (z,z)\} \\ -1 & (O_{t,b}, S_{t,b}) \in \{(1,0), (0,1)\} \\ -\phi & (O_{t,b}, S_{t,b}) \in \{(-x), (x,-), (z,-), (-z), (z,z)\} \end{cases} \]

\[ \text{total}(S, O) = \sum_{t=0}^{k} \sum_{b=0}^{n} \begin{cases} 1 & (O_{t,b}, S_{t,b}) \in \{(0,0), (1,1), (1,0), (0,1)\} \\ \phi & (O_{t,b}, S_{t,b}) \in \{(-x), (x,-), (z,-), (-z), (z,z)\} \end{cases} \]

\[ \text{fitness}(S, O) = \begin{cases} 0 & \text{sum}(S, O) < 0 \\ \frac{\text{sum}(S, O)}{\text{total}(S, O)} & \text{sum}(S, O) \geq 0 \end{cases} \]

- \( x \): uninitialized variable
- \( z \): high impedance
- \( - \): bit value of 0 or 1

\( S_{t,b} \): \( b \)th bit for time \( t \) in output
\( O_{t,b} \): \( b \)th bit for time \( t \) in oracle
Fault Localization

Produces a uniformly ranked set of implicated design code for a faulty circuit description

- AST for circuit design, simulation output, circuit oracle
- Comparison of output wire values between simulation and oracle to get identifier names with output mismatch
- Fixed point analysis of assignments to output wires and registers
- Uniformly ranked set of implicated AST nodes
Fixed Point Analysis of Assignments

**Input:** Faulty circuit design code AST, ast.

**Input:** Output from design simulation, $S : Time \mapsto Var \mapsto \{0, 1, x, z\}$.

**Input:** Oracle for circuit behavior, $O : Time \mapsto Var \mapsto \{0, 1, x, z\}$.

**Output:** Fault localization set, $FL$.

```
1: FL, mismatch ← ∅, ∅
2: mismatch′ ← get_output_mismatch(O, S)
3: while mismatch ≠ mismatch′ do
4:   mismatch ← mismatch ∪ mismatch′
5:   for node in ast do
6:     if implicated(node, mismatch) then
7:       FL ← FL ∪ {node.id}
8:     for each child of node do
9:       FL ← FL ∪ {child.id}
10:    if type(child) = Identifier and
11:        child.name ∉ mismatch then
12:      mismatch′ ← mismatch′ ∪ {child.name}
13: return FL
```

Returns a set of wire/register names that have output mismatch

Two ways to be implicated:
- Assignments: assigned variable in the mismatch set
  (e.g., $\text{count} \leq 1'b1$)
- Conditionals: conditional includes a mismatched variable
  (e.g., $\text{if}(\text{reset}==1'b1) \text{ count} \leq 1'b0$)
More on Methodology in the Paper!

- **Selection**: “choosing parent(s) to produce offspring(s) for the next generation of GP evolution”
- **Repair Operators**: “borrowing code from elsewhere in the parent’s design to produce a child”
- **Repair Templates**: “introducing new design code to the parent to produce a child”
- **Fix Localization**: “guidelines for the APR algorithm to apply edits to design code”
Benchmark Suite of Defect Scenarios

A **defect scenario** consists of:

- A Verilog circuit design
- An instrumented testbench for the design
- A developer-provided oracle for circuit behavior
- A design defect for the circuit
Benchmark Suite of Defect Scenarios

A defect scenario consists of:

- A Verilog circuit design
- An instrumented testbench for the design
- A developer-provided oracle for circuit behavior
- A design defect for the circuit
- A seeded defect by a hardware expert
## Benchmark Suite: Hardware Projects

<table>
<thead>
<tr>
<th>Project</th>
<th>Description</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>decoder_3_to_8</td>
<td>3-to-8 decoder</td>
<td>25</td>
</tr>
<tr>
<td>counter</td>
<td>4-bit counter with an overflow bit</td>
<td>56</td>
</tr>
<tr>
<td>flip_flop</td>
<td>T-flip flop</td>
<td>16</td>
</tr>
<tr>
<td>fsm_full</td>
<td>Finite state machine</td>
<td>115</td>
</tr>
<tr>
<td>lshift_reg</td>
<td>8-bit left shift register</td>
<td>30</td>
</tr>
<tr>
<td>mux_4_1</td>
<td>4-to-1 multiplexer</td>
<td>19</td>
</tr>
<tr>
<td>i2c</td>
<td>Two-wire, bidirectional serial bus for data exchange</td>
<td>2018</td>
</tr>
<tr>
<td>sha3</td>
<td>Cryptographic hash function</td>
<td>499</td>
</tr>
<tr>
<td>tate_pairing</td>
<td>Core for running the Tate bilinear pairing algorithm for elliptic curves</td>
<td>2206</td>
</tr>
<tr>
<td>reed_solomon_decoder</td>
<td>Core for Reed-Solomon error correction</td>
<td>4366</td>
</tr>
<tr>
<td>sdram_controller</td>
<td>Synchronous DRAM (SDRAM) memory controller</td>
<td>420</td>
</tr>
</tbody>
</table>

### Introductory-level VLSI course projects

- i2c
- sha3
- tate_pairing
- sdram_controller

### OpenCores projects

- decoder_3_to_8
- counter
- flip_flop
- fsm_full
- lshift_reg
- mux_4_1
- reed_solomon_decoder

### Open-source GitHub project

- reed_solomon_decoder

---

Experimental Results
Benchmark Suite: Defect Seeding

Recruited three hardware experts to seed defects into circuits

Two categories of defects

- Category 1 (i.e., “easy”)
- Category 2 (i.e., “hard”)

32 defect scenarios in benchmark suite

- 19 Category 1 defects
- 13 Category 2 defects
Experimental Setup

**RQ #1.** What fraction of defect scenarios can CirFix repair?

**RQ #2.** Does CirFix perform better at Category 1 (“easy”) defects compared to Category 2 (“hard”) defects?

**RQ #3.** How effective is the CirFix fitness function at guiding the search a repair? (Spoiler: highly effective; more in the paper!)

**RQ #4.** How sensitive is CirFix to the quality of the information for expected behavior? (Spoiler: not very sensitive; more in the paper!)
RQ #1: Repair Rate for CirFix

CirFix found 21/32 (65.6%) plausible repairs, with 16/32 (50%) deemed to be correct (i.e., high quality) upon manual inspection.

- 2.05 hours average wall-clock time to find a repair.
RQ #1: Repair Rate for CirFix

CirFix found 21/32 (65.6%) plausible repairs, with 16/32 (50%) deemed to be correct (i.e., high quality) upon manual inspection.

- 2.05 hours average wall-clock time to find a repair

Repair rate comparable to strong results from software-based program repair (e.g., GenProg’s 52.4%, Angelix’s 34.1%)
RQ #2: Performance for Individual Defect Categories

CirFix repaired 12 out of 19 (63.2%) Category 1 (i.e., “easy”) defects and 9 out of 13 (69.2%) Category 2 (i.e., “hard”) defects.

- 1.9 hours average wall-clock time to repair Category 1 defects, 2.2 hours average wall-clock time to repair Category 2 defects.
- No evidence of statistically significant difference in the average amount of time to find a repair between Category 1 and 2 defects (two-tailed Mann Whitney U test, $p = 0.374$).

Experimental Results
Conclusion

- CirFix: a framework for automatically repairing defects in hardware designs with a 50% repair rate
- Fitness function based on visibility and comparison
- Fault localization approach based on fixed point analysis of assignments
- First publicly available benchmark for a variety of Verilog defects
  - Replication Materials: [https://github.com/hammad-a/verilog_repair](https://github.com/hammad-a/verilog_repair)

Questions?

Feel free to contact Hammad Ahmad ([hammada@umich.edu](mailto:hammada@umich.edu)) to start a discussion!