DRILLS: Debugging RTL Intelligently with Localization from Long Simulation

Specification mining using waveform dumps from long-running FPGA emulation to localize bugs in complex digital designs.

Donggyu Kim & Vighnesh Iyer

Final Presentation: 5/6/2019

Problem Definition

- Higher RTL design productivity (Chisel, HLS) enables more complex digital designs (e.g. BOOM-v2), but complex designs usually contain subtle bugs
- Tricky bugs only manifest after trillions of cycles (e.g. running SPEC2017), as high-level assertion failures, hanging, or termination with errors
    - e.g. pipeline hung, invalid writeback in ROB (see DESSERT [FPL'18])
- Even with a waveform, it is very laborious or even impossible to figure out where and when the bug originated.
- **Problem**: given many error-free traces and one error containing trace, localize the likely bug location by module or line of RTL and find the fine-grained implicit properties which were violated.

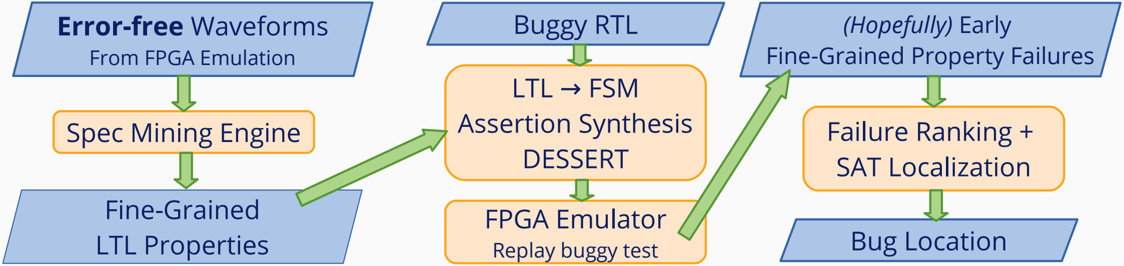
Hypothesis

- *We believe* if a test fails after billions of cycles on a mature RTL design, an assumption the designer made was violated somewhere and at some time
  - The final test failure could have sprung from latent state that was corrupted billions of cycles ago
- The designer's *assumptions* can be extracted from waveforms with 'normal' activity using specification mining
- We can add these mined specifications (as assertions) to the design and replay the failing test to catch a faulty assumption **earlier and with greater locality**

Goals/Approach

- We want a specification mining system that can address:
  - Automated suite of regression assertions
  - Early bug detection and localization on long-running tests
  - Starting point for formal specs of a design
- We aim to first replicate Li's spec mining engine and templates, and apply it to `riscv-mini` on short ISA tests to detect user-inserted bugs
- Next, we will apply the spec mining engine to `rocket-chip` and try it on longer traces

Proposed Approach



LTL (Linear Temporal Logic)

- LTL is a logic for defining *properties* over *traces* of *atomic propositions*
- In the digital design context:
  - A trace is a signal (registers/wires) sampled at rising clock edges; traces are of finite length (not true for LTL in general)
  - Atomic propositions (AP) are functions over signal(s) that evaluate to boolean values

Trace of signal $a = \tau_a = \{0, 1, 1, 1, 0\}$

Trace of signal $b = \tau_b = \{0, 100, 200, 300, 0\}$

AP $x = f(a) = (a == 1)$, AP $y = f(b) = (b >= 200)$

Trace of AP $x = \tau_x = \{0, 1, 1, 1, 0\}$

Trace of AP $y = \tau_y = \{0, 0, 1, 1, 0\}$

LTL Logical Operators

- An LTL formula ϕ is defined over a trace of tuples of APs.
  - Consider the last example: $\tau = \{(), (x), (x, y), (x, y), ()\}$
- An AP on its own is a valid LTL formula
  - For example $\phi = x$. This formula is satisfied if the first element of the trace contains the AP.
- APs can be composed with simple logic operators ($\neg, \wedge, \vee, \rightarrow$) to form another valid LTL formula
  - For example $\phi = (\neg x \vee y)$. This formula is satisfied if the first element of the trace contains the AP.

LTL Temporal Operators

- There are 4 (commonly used) LTL temporal operators (let p, q be APs or LTL formulas):
  - **G**p (*globally*, p must hold for the entire trace)
    $\{(p), (p), (p, q), (p), (p)\}$
  - **F**p (*eventually*, p must eventually hold at some point)
    $\{(), (), (q), (q, p), (p)\}$
  - **X**p (*next*, p must hold in the next timestep)
    $\{(), (p), (p), (q), ()\}$
  - p**U**q (*until*, p must hold until q holds)
    $\{(p), (p), (p, q), (q), ()\}$
- Common combinations include **GF**p (p holds infinitely often) and **FG**p (once p holds, it holds forever)

Hardware Idioms in LTL

Many common temporal RTL patterns are expressible in LTL.

- There should eventually be a response after a request is dispatched
  $\mathbf{G}\,(\mathrm{req} \rightarrow \mathbf{XF}\,\mathrm{resp})$
- The memory bus should respond in 2 cycles
  $\mathbf{G}\,(\mathrm{req} \rightarrow \mathbf{XX}\,\mathrm{resp})$
- `IrrevocableIO` should keep valid high until ready has been asserted
  $\mathbf{G}\,(\mathrm{valid} \rightarrow \mathbf{X}\,(\mathrm{valid}\,\mathbf{U}\,\mathrm{ready}))$
- After a transaction, the slave should be ready again within 2 cycles
  $\mathbf{G}\,((\mathrm{valid} \wedge \mathrm{ready}) \rightarrow (\mathbf{X}\,\mathrm{ready} \vee \mathbf{XX}\,\mathrm{ready}))$

LTL Templates

There are several LTL formulas that can be 'templated' to be filled in with concrete signals from a design.

- Alternating: $a\,\mathbf{A}\,b$
  Example: $\{(a),(a),(b),(a),(b)\}$
- Until: $\mathbf{G}\,(a \rightarrow \mathbf{X}\,(a\,\mathbf{U}\,b))$
- Next: $\mathbf{G}\,(a \rightarrow \mathbf{X}\,b)$
- Eventual: $\mathbf{G}\,(a \rightarrow \mathbf{XF}\,b)$

These property templates are from Wenchao Li's thesis. We implement these templates in the spec mining engine.

Delta Traces

- Signals in RTL designs aren't APs on their own, so we use delta traces to capture *changes* in a signal from one timestep to the next
    - Trace of signal $a = \tau_a = \{0, 1, 1, 1, 0\}$
    - Trace of signal $b = \tau_b = \{0, 100, 200, 300, 0\}$
    - Delta trace of signal $a = \tau_{\Delta a} = \{0, 1, 0, 0, 1\}$
    - Delta trace of signal $b = \tau_{\Delta b} = \{0, 1, 1, 1, 1\}$
- VCD (value change dump) files already store delta traces in compressed format
- Custom user-defined events can be used to construct AP traces (such as AP $f(r, v) = r \wedge v$ for ready/valid interfaces)

Specification Mining

- We use a tool (SPOT) to construct a Buchi-automaton that acts like an LTL property monitor

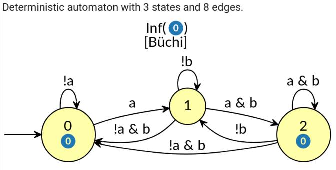


Figure 1: Monitor for $\mathbf{G}\,(a \rightarrow \mathbf{XF}\, b)$

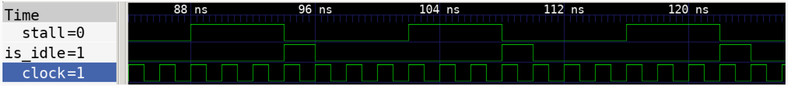
- We use a naive algorithm ($\mathcal{O}(n!)$) to plug in delta traces to the template and check whether the property holds.
  - We use the sparsity of delta traces to perform spec mining efficiently
  - We restrict the signal permutations under consideration to be within 1 module

As of 2 Weeks Ago

- We have a spec miner in Python that can evaluate the ’alternating’ and ’next’ templates using VCD files as input
- It works on `chisel-examples` and `riscv-mini` (and `rocket-chip` if only 1 clock is considered)

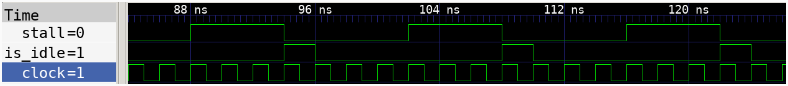
Here’s an example property mined from `riscv-mini`

```
Next: [['TOP.Tile.core.dpath.csr_io_stall',
↪   'TOP.Tile.core.dpath.stall',
↪   'TOP.Tile.core.dpath.csr.io_stall'],
↪   ['TOP.Tile.icache.is_idle']]
```



As of 2 Weeks Ago: Bugs Everywhere!

```
Next: [['TOP.Tile.core.dpath.csr_io_stall',
↪   'TOP.Tile.core.dpath.stall',
↪   'TOP.Tile.core.dpath.csr.io_stall'],
↪   ['TOP.Tile.icache.is_idle']]
```



Look closely, the waveform doesn't match the property template!

Current Status

- Support for all 4 property templates and some tests
- Support for module-level mining and deduplication
- Merging mined properties from several VCDs
- Checking property sets against VCDs
- Notions of falsifiability, falsification, and support
- Automated mine and check flow for `riscv-mini`

Mining on a store ISA test

```
python miner.py --start-time 12 --signal-bit-limit 4 rv32ui-p-sw.vcd
Top 10 properties:
Until TOP.Tile.core.ctrl.io_A_sel -> TOP.Tile.core.ctrl_io_imm_sel, support: 331
Next TOP.Tile.core.dpath.brCond.eq -> TOP.Tile.core.dpath.brCond.neq, support: 291
Next TOP.Tile.core.dpath.brCond.neq -> TOP.Tile.core.dpath.brCond.eq, support: 291
Eventual TOP.Tile.core.ctrl_io_imm_sel -> TOP.Tile.core.dpath.regFile.io_wen, support:
↪  248
Next TOP.Tile.core_io_icache_req_valid -> TOP.Tile.icache.is_idle, support: 233
Next TOP.Tile.core_io_icache_req_valid -> TOP.Tile.icache.is_read, support: 233
Until TOP.Tile.core_io_icache_req_valid -> TOP.Tile.icache.state, support: 233
Until TOP.Tile.icache.is_idle -> TOP.Tile.icache.is_read, support: 233
```

Mining on a jump ISA test

```
python miner.py --start-time 12 --signal-bit-limit 4 rv32ui-p-jal.vcd
Until TOP.Tile.core.dpath.io_ctrl_A_sel -> TOP.Tile.core.dpath_io_ctrl_imm_sel,
↪  support: 37
Until TOP.Tile.core.ctrl.io_wb_sel -> TOP.Tile.core.dpath_io_ctrl_imm_sel, support: 34
Next TOP.Tile.core.dpath_io_ctrl_inst_kill -> TOP.Tile.core.ctrl_io_pc_sel, support: 33
Next TOP.Tile.core.dpath_io_ctrl_inst_kill -> TOP.Tile.core.ctrl.io_wb_sel, support: 33
Next TOP.Tile.core.ctrl_io_pc_sel -> TOP.Tile.core.dpath_io_ctrl_inst_kill, support: 33
Next TOP.Tile.core.ctrl_io_pc_sel -> TOP.Tile.core.ctrl.io_wb_sel, support: 33
Next TOP.Tile.core.dpath.brCond.eq -> TOP.Tile.core.dpath.brCond.neq, support: 33
Next TOP.Tile.core.dpath.brCond.neq -> TOP.Tile.core.dpath.brCond.eq, support: 33
```

Bug Localization

Can typos or copy/paste mistakes be caught?

```diff
  io.csr_cmd := ctrlSignals(11)
- io.illegal := ctrlSignals(12)
+ io.illegal := ctrlSignals(11)
```

This caused a test to hang, so let's check if any mined specs were violated:

```
python checker.py --start-time 12 --signal-bit-limit 4 rv32ui-p-sub.vcd
↪  riscv_mini.props
ERROR on property Until TOP.Tile.core.dpath.csr.io_illegal ->
↪  TOP.Tile.icache.io_cpu_req_valid
ERROR on property Until TOP.Tile.core.dpath.csr.io_illegal ->
↪  TOP.Tile.icache.io_cpu_resp_valid
ERROR on property Until TOP.Tile.core.dpath.csr.io_illegal ->
↪  TOP.Tile.core.ctrl.io_A_sel
```

Challenges

- Many of the specs are meaningless or require very specific behaviors to falsify
- Some of the specs can be statically inferred (like shift registers)
- These spec templates aren’t sufficient to catch some typo bugs
- The VCD parser is too slow to run on longer traces like the RISC-V benchmarks

Current Status and Future Work

- We built a spec mining engine to replicate the prior work
- We applied the engine to `riscv-mini` and demonstrated its strengths and weaknesses
- There's a lot of future work in refining the spec templates and in software engineering
- The ultimate goal of applying this work to debug BOOM is a while away