Coverage, Data Mining & Machine Learning in Verification

Stan Sokorac
Director of Engineering
Tenstorrent, Inc.

March 29, 2018

About Me

- Graduated from University of Toronto
- Worked at IBM, ATI, AMD, ARM
- Worked as Design Engineer, Verification Engineer, Technical Lead, Sr. Manager
- Currently developing a high-performance CPU designed for deep learning

 Generated using Dreamscape (dreamscopeapp.com)
Agenda

- Coverage
  - Why?
  - What?
  - Code Coverage
  - Functional Coverage
  - Statistical Coverage / Data Mining

- Machine Learning
  - Why ML in verification?
  - DVCON 2017 Paper:
    - Using Machine Learning to Optimize Random Test Constraints

- Closing Coverage
  - When are we done?
  - Challenges

Why do we need coverage?

- Constrained random stimulus
  - Throw a bunch of things at the design, see what sticks

- Hard to tell what's going on
  - Failing tests means something good is going on
  - Passing tests don't mean much unless we have a way to confirm that they are doing something
  - Looking at waveforms can be misleading

- Hard to tell when to stop
  - Am I not finding bugs because there aren't any, or because my tests are bad?

- Coverage is the best way to get that feedback
What is coverage?

- Coverage is a way to tell to what degree a piece of design should be and has been tested (“covered”)

Types of coverage
- Code Coverage (also used in computer science)
- Functional Coverage
- Statistical Coverage

Code Coverage

- The simplest and easiest to gather
- Great way to identify major holes in stimulus
- Very bad at telling you that you're done

- Many types:
  - Statement
  - Branch
  - Expression
  - FSM
  - Toggle
  - I/O
Statement Coverage

- The most straightforward and simple
- Each logic statement is one coverage bin:

Challenges:
- Dead code! Error handling code?

```verilog
module my_logic (input clk, input a, output b);
    wire c;
    assign b = ~c;
endmodule
```

Branch Coverage

- Two (or more!) possible results on every branch
- Goes a bit deeper into the design, harder to hit, but also harder to analyze

```verilog
module my_logic(
    input clk,
    input a, b,
    output c, d);
    reg c, e;
    assign d = ~a;
    always @(posedge clk) begin
        c <= a ? b : d;
        if (a & b)
            e <= d;
        else
            e <= 1'b1;
    end
endmodule
```

Case-

```verilog
case(state) IDLE: if (req) nextState <= REQ;
    REQ: if (ack) nextState <= IDLE;
    default: nextState <= 'bX;
endcase
```

Condition

- For each branch, coverage will show the true and false outcome separately
- Can have significant overlap with statement coverage, depending on how code is written
Expression Coverage

- Logic expression can reach its result many different ways:

```
if (a & b)  
   true: a = 1, b = 1
   false: a = 0, b = 1
   a = 1, b = 0
   a = 0, b = 0
```

- Even deeper, even harder to close, but can provide very interesting feedback
- However, many points are irrelevant, some even impossible to hit
  - Very time consuming – usually not the best way to spend your (limited!) time

Other code coverage

- Different vendors provide different options for additional coverage
- Toggle coverage examines each signal’s toggling
- Toggle I/O will ensure that all ports have toggled
- FSM coverage recognized state machines and checks state transitions
Functional Coverage

- Code coverage isn’t enough
  - It doesn’t cover code that isn’t written
  - It can’t guess what all legal values are
  - Did the results even matter?
  - Could you detect an error if the line was wrong?
  - It doesn’t capture context of events

- Functional coverage provides more focused data
  - driven by specifications (features) and experience (areas of concern)

- Key differences
  - Hand-written, custom coverage points – more likely to be highly relevant
  - Takes into account any combination of events
  - Easier to analyze and understand

Functional Coverage on RTL

- Same syntax as SVA assertions, but with ‘cover’ instead of ‘assert’:

```vhdl
// Every request must have an ack within two cycles
ack_must_come: assert property (@(posedge clk) req => ##[1:2] ack);

// We’ve seen a request with ack on the next cycle
ack_after_one: cover property (@(posedge clk) req => ##1 ack);

// We’ve seen a request with ack two cycles later
ack_after_two: cover property (@(posedge clk) req => ##2 ack);
```

- Commonly used for:
  - Interface coverage
  - Internal structures
  - Areas of concern identified by RTL designers
Functional Coverage in TB

- Testbench components built to track high-level behaviour
- Writing high-level coverage much easier at that level:

```vhdl
// A read and write requests on interface - covergroup
covergroup intf_requests with function sample(txn req);
  bins read = req.is_read_type();  // generates two bins : 0 and 1
  bins write = req.is_write_type();
endgroup

// A read and write requests on interface - cover property
ingf_read: cover property (@(posedge clk)
  (req_v & (@(req_type == `REQ_READ_CLEAN) |
             (req_type == `REQ_READ_UNIQUE) |
             (req_type == `READ_READ_NO_SNOOP) )));

intf_write: cover property (@(posedge clk)
  (req_v & (@(req_type == `REQ_WRITE_PARTIAL) |
             (req_type == `REQ_WRITE_FULL) |
             (req_type == `REQ_WRITE_NO_SNOOP) )));
```

Functional Coverage Crosses

- Easy combination of coverage bins:

```vhdl
covergroup intf_requests with function sample(txn req);
  bins req_type = req.req_type();
  bins size = req.size();
  // All request types in all sizes
  cross req_type, size;
endgroup
cross read, size, current_state, fifo_count;
```

- PRO: One-liner can provide many points
- CON: One-liner can provide too many points
  - Just because you can write it easily, doesn’t mean you should!
Statistical Coverage – Data Mining

- New concept pioneered by ARM, with some EDA adoption already
- Digging deeper into coverage data
- Present large quantities of data in an easy-to-understand way

Statistical Coverage – Example 1
Statistical Coverage – Example 2

- Number of outstanding requests on an interface targeted by two different instruction generators

Closing Coverage

- The most important question for a verification engineer:  
  - Are we done?

- Coverage doesn’t answer this definitively (nothing really does 😊 ), but it helps tremendously
  - It can definitively say ‘no’, at least

- What does it mean for a coverage point to be hit?  
  - Hit once  
  - Hit many times – how many is enough?
Closing Coverage - Challenges

- The coverage closing loop:
  - Coverage collection
  - Analysis
  - Identifying missed points
  - Waiving of impossible or uninteresting points
  - Stimulus adjustment

- Analysis of too many points is time consuming

- Stimulus adjustments can be difficult:
  - Easy to add a missing request type
  - Hard to hit a deep coverage points on an internal RTL structure

Closing Coverage - Efficiency

- Constrained random often means indirect constraint adjustment and millions or billions of cycles of “hoping to hit” new points

- First few hit the most, after that we’re wasting a lot of time

- Very expensive – both in machine time, and project time

- Need new solutions and ideas!
  - Formal, Machine Learning, ???
Machine Learning & Data Mining

- “Machine learning is a type of artificial intelligence (AI) that provides computers with the ability to learn without being explicitly programmed. Machine learning focuses on the development of computer programs that can change when exposed to new data. The process of machine learning is similar to that of data mining. Both systems search through data to look for patterns. However, instead of extracting data for human comprehension – as is the case in data mining applications – machine learning uses that data to detect patterns in data and adjust program actions accordingly.” (whatis.techtarget.com)

Machine Learning & Data Mining in Verification

- Good fit with challenges in verification, especially with coverage closing and improving efficiency

- Coverage can generate large amounts of data
- Billions of simulation cycles is too much data for humans to process directly

- Most efforts in ML applications to Verification involved coverage:
  - Hitting more (all!) coverage more quickly
  - Using coverage data to find the most efficient tests
  - Using coverage data to find the bugs earlier
Predicting Coverage Using Neural Networks

- A simple densely connected NN can be trained to predict test coverage results
- An example on the right shows that it can be done to a great degree of accuracy
- Since test “knobs” are usually limited to a few hundred, training can be done any computer

Sequence Predictions using LSTMs

- LSTM is a type of network that can learn and predict sequence attributes
- Most random tests are sequences of transactions!
- LSTM can be trained to predict design behavior from an input stream…
- ...or, to tell us what transaction to send next to get desired events!
Fun Example

- The Unreasonable Effectiveness of Recurrent Neural Networks

PANDARUS:
Alas, I think he shall be come approached and the day
When little strain would be attain’d into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Second Senator:
They are away this miseries, produced upon my soul,
Breaking and strongly should be buried, when I perish
The earth and thoughts of many states.

DUKE VINCENTIO:
Well, your wit is in the care of side and that.

Possible Applications

1. Optimize test lists to better target desired areas
   - Find bugs in recently changed code, find bugs in tricky code
   - Focus on traditionally bug-filled areas
   - Close coverage faster, hit coverage that’s hard to hit
   - Target poorly covered areas of the design
2. Create new generation of stimulus generators
3. Analyze RTL and SV code to predict possible trouble areas
4. Analyze waveforms to find exceptions, triage problems
Questions?

Optimizing Random Test Constraints Using Machine Learning Algorithms

Stan Sokorac
stan.sokorac@arm.com

Department of Electrical and Computer Engineering, The University of Texas at Austin
Stan Sokorac, Tenstorrent, March 29, 2018
Background

- Modern designs are extremely complex
  - Impossible to come up with every possible combination of stimulus by hand
- Constrained random simulation is a staple of verification
  - Generation of random instruction streams controlled through a set of adjustable constraints
  - Great at hitting many common and uncommon design corners

However, random testing is also inefficient and expensive!
- Random distributions hit most common cases most often, spending majority of the time testing the same things over and over
  - Hard to find bugs take a long time to find!
The Testing Loop

- A new type of coverage
  - A way to extract information about a single test, to provide feedback on its quality

- A way to use this feedback in machine learning algorithms
  - Optimization designed to find hard to find bugs quicker

Finding hard-to-find bugs

- Non-trivial bugs require a combination of events and state changes to occur in close proximity
- Most bugs aren’t particularly “deep”
  - It takes a couple of things to line up that we usually haven’t thought to line up

- Verification engineers bias stimulus towards areas that are likely to cause bugs
  - Great use of experience and knowledge to find most bugs
  - However, we can’t just keep running the same things

- Need an objective way to evaluate test variety and coverage
  - Objective is the key – we must eliminate the bias from hand-written functional coverage to find the hard-to-find corner cases
Exploring the state space

- One objective view of design coverage is its state space
  - State space of the design is represented by all of its flops
  - The total space size is $2^{\text{flops}}$, which is not practical to track
  - The interesting things happen when state changes
    - Flop toggle coverage – good start, but too simple, like CCOV

Lining things up

- Approximation for “events lining up” that takes design state into account:
  - Two flops toggling in close proximity in time
  - Still fairly simple to track (state space is flops$^2$), but much more interesting than single flop toggle
  - Very objective – requires no understanding of the design
Toggle matrix

- Yellow represent areas of high toggle counts, red are low, and white are blank.
- Logarithmic scale – yellows are an order of magnitude higher than reds.
- This represents one randomly picked test.

Interpreting the results

- How many total toggle pairs a test produces:
  - indication of the volume of activity

- How many toggle pairs (bins) are exercised by the test:
  - indication of the breadth of the test

- We also need to focus on hard to hit bins that are rarely exercised
  - Don’t bother optimizing for bins that are hit all the time
  - Filter anything that is easy to hit – bins hit by more than 50% of the tests is a good start
Scoring a Test

- Having a “score” for a test good for learning algorithms
- High score means:
  - High activity of rare events in the test (volume)
  - Many different rare events hit (breadth)
- Then, we calculate the score:

\[
\text{Score} = \frac{\text{FilteredVolume}^2 + \text{Rare\_Factor} \times \text{FilteredBreadth}^2}{\text{Power\_Factor}}
\]

- Rare\_Factor / Power\_Factor provide easy tuning

Machine Learning through a Genetic Algorithm

- A type of reinforced learning algorithm
- Select a random population of tests, and evaluate each
- Create the next generation of tests by:
  - mutating (slightly adjusting constraints) current tests
  - mating (take an average of two tests) current tests
- The evaluation score dictates the chance of a test participating in the next generation

- Toggle pair coverage score used to select tests
Iteration Performance

- Progress is charted through each iteration
- The iterations of interest are the ones that:
  - Show spikes over previous iterations
  - Show overall highest averages or totals
  - Have exposed new fail signatures

- It's important to monitor number of new bins hit, as well as bins "lost", i.e. bins that we no longer hit in the latest iteration (see above)

Volume vs Breadth over iterations
Does it find bugs?

- Yes! It’s still early, but the data is promising on LSU and L2
  - One of the iterations found a new bug, optimized large run found 3 more and failed over 450 times

<table>
<thead>
<tr>
<th>Regression</th>
<th>Test Count</th>
<th>Fail Count</th>
<th>Pass Rate</th>
<th>Cycles</th>
<th>Unique Signatures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular weekly run</td>
<td>30000</td>
<td>24</td>
<td>99.92</td>
<td>173.6 Million</td>
<td>4</td>
</tr>
<tr>
<td>6 iterations of 500 tests</td>
<td>2749</td>
<td>41</td>
<td>98.5</td>
<td>15.4 Million</td>
<td>5</td>
</tr>
<tr>
<td>Large run using 6th iteration test selection</td>
<td>30000</td>
<td>469</td>
<td>98.43</td>
<td>166.1 Million</td>
<td>8</td>
</tr>
</tbody>
</table>

Other ML algorithms – NNs and SVMs

- Genetic algorithms require feedback on each test, making iterations slow
- Training a model such as a neural network to predict scores would speed up this loop
Other ML algorithms – Unsupervised Learning

- A clustering algorithm can detect groups of test that are “similar”
  - This can be used to “spread” the tests around
  - Run separate optimization on each cluster

- Anomaly detection
  - Algorithm that detects tests that are significant
  - This kind of a test is more likely to hit new corner cases

Where To Go From Here?

- This work is in early stages, and there are many ideas and trials to go through!

- Try other projects and designs
- Use meta-learning to learn the best GA parameters
- Continue to experiment with other ML algorithms
Questions?