Coverage Discounting: A Generalized Approach for Testbench Qualification

Peter Lisherness and Kwang-Ting (Tim) Cheng
University of California, Santa Barbara
Email: {peter,timcheng}@ece.ucsb.edu

***Abstract*—In simulation-based validation, the detection of design errors requires both stimulus capable of activating the errors and checkers capable of detecting the behavior as erroneous. Validation coverage metrics tend to address only the sufficiency of a testbench's stimulus component, whereas fault insertion techniques focus on the testbench's checker component. In this paper we introduce "coverage discounting", an analytical technique that combines the benefits of each approach, overcomes their respective shortcomings, and provides significantly more information than performing both tasks separately. The proposed approach can be used with any functional coverage metric (including, and ideally, user defined covergroups and bins), and a variety of fault models and insertion mechanisms. We present an experimental case study where the proposed approach is used to evaluate functional and pseudofunctional tests for a microprocessor. The simulation efficiency is improved through the use of an instruction set simulator, which has been instrumented to record functional coverage information as well as insert faults according to an ad-hoc fault model. The results demonstrate the benefits of coverage discounting: it is able to correctly distinguish high- and low-quality tests with similar coverage scores as well as expose checker insufficiencies.**

I. Introduction

Coverage metrics are almost universally employed in simulation-based hardware validation as a measure of test thoroughness: they provide confidence that the design's functionality has been thoroughly exercised or, if it hasn't, direct the validation engineers toward "holes": those uncovered areas that need additional attention [1]. The key contribution of this paper is **coverage discounting**, a method that revises the coverage score downward by identifying coverpoints that were only vacuously covered and require more careful testing. In this study, we demonstrate this technique with an instruction set simulator that has been instrumented to collect functional coverage data and perform fault simulation with an ad-hoc fault model targeted at the functional coverpoints.

The most common validation coverage metrics include implicit metrics, such as code, path, and toggle coverage, and specification derived functional metrics such as assertions and coverpoints (ex. bins and sequences) [2]. Unfortunately, these implicit and functional metrics are both indifferent to the quality of the testbench's checker: they determine whether certain states have been reached, but not whether the tests can actually detect any errors those states may cause. Fault insertion, such as mutation analysis [3], has been proposed to evaluate the tests' ability to detect errors, originally for software testing. While this technique has been adapted to hardware validation tools [4]–[7], the information it provides is of limited utility: it does not expose test holes with a specific functional meaning (as coverage metrics do). The holes provided by fault insertion are undetected faults, but these faults are neither guaranteed detectable at all nor are they necessarily functionally meaningful.

Coverage discounting is an analytical method that, for any given implicit or functional coverage metric and any given fault model or injection mechanism, extracts meaningful coverage holes from the meaningless or misleading fault insertion results. The basic premise is as follows:

1) An inserted fault may change the design's functionality.
2) This change may result in a different coverage score.
3) If the fault is not detected, then those covered points which are no longer covered in the presence of the fault should not be considered covered in the fault-free design either.

So the detection information from fault insertion is mapped back onto the coverage metric, and any undetected coverpoints are removed from the "covered" set. After multiple fault insertions, the revised coverage score will better reflect the portion of the coverpoints that were meaningfully tested. The removed coverpoints were only vacuously activated, and still represent test holes that need to be more carefully targeted.

The proposed technique can be applied to any activation-based coverage metric, including implicit and functional metrics, and the fault detection portion can use any mixture of mutation, perturbation, or ad-hoc fault insertion methods. It is adaptable to any tool, design language, or simulation environment that is capable of producing coverage and fault detection data; all that is needed is the ability to parse and manipulate that data.

II. Coverage Discounting - An Example

In this section, we will walk through a simplified example of how coverage discounting works for a single coverpoint. For ease of explanation we are using statement coverage as the coverage metric and mutation analysis as the fault model. Note that a different metric and fault model are used in the experimental results of Sec. V: as we will explain in later sections, statement coverage and mutation analysis aren't necessarily the best choices in a real validation environment.

The following listing shows a block of code (the design under test) with a built-in assertion checker:

978-1-4577-1743-7/11/$26.00 ©2011 IEEE

Listing 1. Example Design

```
1 //Test Vectors: a1=2, a2=1, a3=0
2 input a; output b;
3 if (a>=0)
4 ⇒ b=1+a;
5 else
6    b=1-a;
7 //Test Outputs: b1=3, b2=2, b3=1
8 assert(b!=0); //The checker
```

After simulation of vector a_1=2, the code coverage report shows that line 4 is covered (by the statement coverage metric). We will next run a fault insertion campaign using mutation analysis. One of the inserted faults changes the code to the following:

Listing 2. Example Design with Mutant

```
1 //Test Vectors: a1=2, a2=1, a3=0
2 input a; output b;
3 if (a<=0) //Fault, orig. (a>=0)
4 ⇏ b=1+a;
5 else
6    b=1-a;
7 //Test Outputs: b1=-1, b2=0, b3=1
8 assert(b!=0); //The checker
```

Here, the inequality on line 3 has been switched from ">=" to "<=". Simulation of this fault with the test vector a_1=2 and checker `assert(b!=0)` will pass, seeing as the checker is insufficient to detect the result b_1=−1 as erroneous. Simulating only this test vector, the fault is undetected.

Also, the code coverage report has changed: line 4 was not covered in the presence of the fault. From this we conclude that the stimulus and checker were not sufficient to thoroughly cover line 4, so it is removed from vector a_1's coverage score. This is the essence of coverage discounting: we reduce coverage scores (in this case removing line 4 from a_1's covered set) to reflect the validation holes revealed by undetected faults.

A. Fixing the Hole

Although we determined that a_1 did not meaningfully cover that statement, some other stimulus in the test set may. For instance, the test vector a_2=1 would cover that statement *and* allow the checker to detect the fault. Similarly, a more complete checker, such as `assert(b==abs(a)+1)`, would allow even the original vector a_1 to cover the statement.

B. Redundant Faults

The primary problem with mutation analysis is its tendency to create equivalent mutants, a form of redundant fault that is by definition not testable. Although some can be automatically classified as redundant, the rest can require enormous manual effort to diagnose.

The listing below shows an example of an equivalent mutant:

Listing 3. Example Design with Equivalent Mutant

```
1 //Test Vectors: a1=2, a2=1, a3=0
2 input a; output b;
3 if (a>0) //Fault, orig. (a>=0)
4 ⇒ b=1+a;
5 else
6    b=1-a;
7 //Test Outputs: b1=3, b2=2, b3=1
8 assert(b!=0); //The checker
```

The fault on line 3 does not change the function of the code, because when $a = 0$ (the only input affected), both $b = 1 + a$ and $b = 1 - a$ evaluate to $b = 1$. When performing fault insertion alone, unless identified as a redundant fault, this undetected fault would be considered a coverage hole.

Let us examine the behavior of coverage discounting in this scenario. The test vector a_3=0 will cover line 4 normally, but not in the presence of the fault. If the vector a_3=0 is the only vector in the test set, line 4 will not be considered covered. This is a fair assessment: under that vector, there is a redundant code path in the original, unmodified code.

But this statement coverage hole can be filled by other vectors, unlike a redundant fault: a different test vector that exercises line 4, such as the vector a_2=1, can cover this statement. So even though the fault is untestable, any holes it reopens when we discount the coverage score can still be covered. This is one of the primary benefits of coverage discounting: redundant faults do not need to be identified, as they are either ignored or expose fundamental redundancies in the underlying implementation.

III. Background and Related Work

In this section, we detail existing validation coverage metrics and fault insertion methods. A primary contribution of this work is that it unites these two previously separate validation practices. Therefore, the following is intended to serve as a survey of the methods that can be paired for coverage discounting, while also identifying the shortcomings of each without our proposed approach.

A. Validation Coverage Metrics

This subsection details those coverage metrics most commonly used in simulation-based hardware validation, and should not be confused with fault coverage as is used in manufacturing test. A discussion on fault coverage is included later in Sec. III-B. For a more complete and in-depth survey of coverage metrics in validation, refer to [8].

1) Implicit (Code/Structural) Coverage: Implicit coverage metrics are automatically derived from, and implied by, the design. In particular, RTL statement, branch, and path coverage metrics all evaluate the thoroughness with which the RTL code's control flow graph (CFG) has been traversed. Similarly, toggle coverage measures how many signals in an RT- or gate-level description have undergone some dynamic switching.

There are two primary advantages to implicit metrics. First, they are objective: because they are not manually defined, they

aren't inherently biased by what a designer or validation engineer thinks is important. Secondly, they are easy to measure. Support for these metrics is built in to most simulators and compilers, and little manual effort is required to enable it.

Despite these benefits, there are many problems with implicit metrics:

- They are highly sensitive to code style and structure (particularly the CFG-oriented metrics).
- They tend to be coarse-grained (with the exception of path coverage), so even a woefully inadequate test set can often reach 100%.
- Because they only consider what is implemented, they are not suited for uncovering errors of omission.
- The coverage holes can be difficult to analyze, as the validation engineer must gain an intimate knowledge of the design implementation to determine how to write appropriate tests.
- These coverage metrics must be computed in the abstraction level they are derived from. That is, RTL code coverage cannot be computed through architectural simulation of the tests, gate-level toggle coverage cannot be computed without gate-level simulation, etc.
- They only measure activation and are unable to evaluate if activated faulty behavior could indeed be detected by the testbench as erroneous.

While implicit metrics are still an important part of validation, due to these shortcomings they are usually used in combination with functional coverage metrics.

2) Functional Coverage: Specification-derived functional coverpoints have become the mainstay of modern simulation-based validation [1], [8]. These coverpoints are essentially event triggers attached to various behaviors defined in the functional specification. For example, a microprocessor may have coverpoints defined for each opcode, or an arbiter might have every possible request/grant sequence encoded as an individual coverpoint. Infeasible or invalid events can be defined as such, acting as assertions. Each coverpoint or group thereof can have an individual weight assigned to it based on importance, so the final coverage score depends more heavily on the "important" specification aspects than the "unimportant" ones. A coverage threshold can also be set for each coverpoint, requiring that it be triggered, or "hit", a certain number of times before it is considered covered. When unspecified, this threshold is usually assumed to be 1 (that is, a single hit is sufficient to cover the coverpoint).

Functional coverage metrics overcome many of the problems with implicit metrics because they are based on the specification rather than the design implementation:

- They are generally specified separately from the design implementation and thus insensitive to coding style.
- Their granularity depends entirely on how many coverpoints the validation engineers choose to generate.
- They encode the specification, and this can help catch errors of omission in the implementation.
- The coverage holes have a specific functional meaning - a validation engineer can often target them without knowledge of the implementation.
- They do not always need to be computed at a particular level of abstraction. For example, an architectural simulator can be used to compute some functional coverage metrics for a post-silicon validation test.

With these benefits, they lose the objectivity and simplicity of implicit metrics. Although it is possible to automatically generate some functional coverpoints from machine-parsable functional specifications, today the bulk of these coverpoints are generated manually (as complete parsable specifications are rarely available). Functional coverage metrics also share the largest shortcoming of implicit metrics: they still only measure activation and are oblivious to the sufficiency or insufficiency of the testbench's checkers.

3) Observability Coverage: Observability coverage metrics were first proposed in [9] to address the activation-only nature of other validation coverage metrics. Since then, multiple studies have been done attempting to generalize or extend the technique [10] [11]. Observability coverage metrics are a form of implicit metric which adds data-flow evaluation. Statements are considered covered only if they are executed *and* the result of that execution has a dynamic data flow to the output, where it is assumed to be checked.

There are some problems with observability coverage metrics that cannot be easily solved in their current formulation:

- They require specialized simulation tools with extensive instrumentation to record the dynamic dataflows. This incurs significant overhead and limits the use of these metrics to supported platforms/languages.
- The observability concept only works with implicit metrics; it cannot be readily combined with arbitrary functional coverpoints.
- The dataflow calculus is approximate and can produce incorrect results in the presence of masking or reconvergent fanouts.
- The presence and quality of the checker are assumed, but never actually evaluated.

The last problem is the most troubling, it means that observability coverage does not really solve the problem it ostensibly sets out to solve. Consider the example from Sec. II: observability coverage would have been no better than statement coverage because the data *does* propagate to an output/checker, but the checker itself is incomplete and therefore unable to detect the error.

Our proposed approach overcomes all of these problems:

- No specialized simulation infrastructure is needed, only the ability to insert faults and record coverage. These are both common tasks with widespread support.
- Coverage discounting can be performed on any mix of arbitrary functional coverpoints.
- Coverage discounting relies on explicit fault injection, which behaves correctly in the presence of masking or reconvergent fanouts.
- The presence and quality of the checker are directly

evaluated in coverage discounting: only when faults elude detection by the checker are they used to revise the coverage scores.

B. Fault Insertion

Fault insertion in validation serves two purposes:

1) Fault coverage is an implicit coverage metric [12].
2) Fault detection demonstrates the relative quality of the testbench checkers.

Fault coverage can be problematic when used as a validation coverage metric. In particular, the analysis of the fault coverage holes can be very difficult; not only is the validation engineer required to gain an intimate understanding of the implementation, they further need to analyze how the design behaves in the presence of each undetected fault. If the fault is redundant, this is a complete waste of effort: no additional tests can be written to cover it, and no additional insight about any *actual* design errors is gained. Fault coverage also lacks one of the primary benefits of other implicit (ex. code coverage) metrics: it is generally very expensive to compute.

The other use of fault insertion in validation, evaluating checker quality, suffers the same redundant fault problem: no additional functional checkers can detect these faults. It may be possible to detect a functionally redundant fault by, for example, checking internal signals. However, the resulting checker would be an artifact of the implementation rather than the functional specification, and as such would be unlikely to detect any actual implementation errors.

On the opposite end of the spectrum from redundant faults, easily detected faults can also be troublesome. Undetected faults are the primary source of information, so a fault model which produces many large, catastrophic faults that are trivially detected provides little insight into the quality of the test stimulus or checkers. Therefore, there is motivation to select a fault model that is "subtle", minimizing the number of detected faults and, in turn, improving the chance of extracting useful information out of each fault simulated.

There is an inherent trade-off between redundant faults and trivially detectable faults. In [13] and [14] the authors determine that the "best" fault models, i.e. those that produce subtle and difficult to detect errors, are also those most likely to produce redundant faults.

Trivially detectable faults are not directly addressed by our approach: they still present a simulation overhead. However, by effectively ignoring redundant faults (that is, not treating them as coverage holes), coverage discounting enables the end user to employ more subtle fault models without the overhead of redundant fault identification.

1) Mutation Analysis: As previously mentioned, mutation analysis is a well established software testing technique [3] which has recently been adopted for hardware validation [4]–[7]. An overview of mutation analysis research can be found in [15]. The basic premise of mutation analysis is as follows:

1) A syntactic change is made to the design/program under test, for example changing the value of a constant, inverting an inequality, or deleting a unary operator. This modified version of the DUT is called a "mutant".
2) The validation tests are applied to the mutant.
3) If any of these tests fails (i.e. detects the fault), the mutant is classified as "killed".
4) Steps 1-3 are repeated for every candidate fault (collectively known as the mutant library).
5) Any mutants not killed are analyzed: if they are untestable they are classified as "equivalent".
6) Additional testcases are written until all mutants are either "killed" or "equivalent".

At the end of this process, the test set is considered "mutation complete". There are some variations on this theme, such as weak mutation [16], which reduces mutation analysis to an activation-only implicit coverage metric, or selective mutation [17], which limits the mutation library to a specific subset of fault types. There has also been an enormous amount of work in automatically detecting equivalent mutants, both via static [18] and dynamic [19] analysis. Despite these efforts, automatic classification of all equivalent mutants remains an open problem.

The commercial HDL mutation analysis tool, Certitude [5], attempts to avoid the need for equivalence analysis by only reporting those mutants which cause the DUT to produce different outputs at module boundaries. This is tantamount to automatically inserting additional validation checkers, and determining whether the checkers provided by the original testbench are as comprehensive. There are two problems with this approach:

1) It ignores those mutants whose effects do not propagate to outputs which but are nevertheless not equivalent.
2) Even if the output at module boundaries differs, the mutant is not guaranteed non-equivalent. It is possible that the difference is not functionally meaningful (ex. the error may only affect "don't care" bits).

Despite these shortcomings, it should be possible to use the Certitude tool for the fault insertion portion of coverage discounting.

IV. Details of Proposed Approach

In this section we discuss the theoretical basis of coverage discounting, as well as the practical and scalability considerations in selecting an appropriate coverage metric, fault model, and simulation environment.

A. Theoretical Basis

The design under test (DUT) can be seen as a large finite state machine (FSM), with the validation test stimulus being an interacting FSM that, when combined with the design under test, produces an autonomous FSM (i.e. one having no inputs). The assertions and other checkers in the testbench watch this FSM for invalid states or sequences, and produce a "pass" or "fail" verdict based on whether or not the DUT's FSM triggers any of this invalid behavior. The functional coverpoints also watch the DUT FSM, with each coverpoint corresponding to some set of states or sequences. For example, a coverpoint of

"opcode=add" for a microprocessor will match any machine state where the instruction register matches the add opcode. Whenever the state matches a coverpoint, a counter associated with that coverpoint is incremented. Usually, any nonzero count in this counter is enough for that coverpoint to be considered "covered".

When we insert a non-redundant fault, it changes the DUT's FSM, replacing one or more of the state transition edges. A test can be said to "activate" this fault if it traverses one or more of these modified edges. As soon as this happens, the set of states entered over the course of the test will change. This may eventually lead the DUT's FSM to enter an invalid state, which the assertions and checkers will detect and produce a "fail" verdict. It also may not - the FSM may stay in valid states, even in the presence of the fault, and end with a "pass" result. If the result is "pass", it could be that either the fault did not substantively change the behavior of the DUT or that the checker and/or assertions do not encode the full set of invalid behaviors.

Just as the checkers and assertions are watching this new state trajectory, so are the coverpoint monitors. As a different set of states is passed through, these monitors may be triggered at different times or in different amounts than they were by the fault-free FSM. Some coverpoints may go from being covered in the fault-free FSM to uncovered in the presence of the fault. If the fault is detected, there is little that can be said about these coverpoints. If, however, the fault is not detected, it implies that the portion of the fault-free state trajectory that triggered those coverpoints was not a necessary condition for satisfying the checkers and assertions.

B. General Method

The iterative flow for computing a discounted coverage score takes the following steps:

1) A fault-free simulation of the DUT and validation tests is performed, and the resulting coverage report (representing the set of covered coverpoints) is used to initialize the working covered set $\mathcal{C}$.
2) A fault is inserted into the DUT and simulated with the tests, producing a report of the "faulty" covered set $\mathcal{F}$.
3) If the fault is not detected by the tests, the working coverage report is replaced by its intersection with the faulty coverage: $\mathcal{C} \leftarrow \mathcal{C} \cap \mathcal{F}$. This removes from $\mathcal{C}$ all coverpoints that were not also in $\mathcal{F}$.
4) Steps 2 and 3 are repeated until the list of faults is exhausted.
5) The covered set $\mathcal{C}$ is used to compute the final discounted coverage score.

C. Coverage Metric Considerations

Because the coverpoints are treated by the proposed approach are a generic set, any coverage metric can be discounted. Moreover, multiple coverage metrics can be combined and discounted simultaneously. Despite this lack of restriction, there are a few practical concerns that make some coverage metrics better than others.

Functional coverage metrics are a better choice for discounting than implicit metrics for a number of reasons. First, the resulting holes are more readily analyzed by validation engineers. Second, they can often be simulated in more abstract models, which is generally more computationally efficient (more on this in Sec. IV-E). Finally, they tend to allow for finer granularity, which can affect the ability of the discounting process to converge quickly.

To understand this last point, consider that the coverpoints are each tied to counters, and any count greater than the threshold at the end of a test is considered "covered". Coarse grained coverage metrics tend to be easier to trigger and therefore end with higher counts. Even if a fault suppresses *some* of these triggers, the coverpoint cannot be discounted unless enough of them are suppressed to lower the count below the threshold (which is often 1, meaning *all* of the hits need to be suppressed).

High granularity metrics where each coverpoint is triggered only rarely should therefore have more coverage holes uncovered by discounting. Similarly, coverpoints with high thresholds will be more readily discounted than those with low thresholds. Both of these concerns relate to holes even without coverage discounting: high granularity and high thresholds tend to generate more holes to begin with.

Scalability is another concern in the selection of a coverage metric. Some overhead is imposed by the instrumentation necessary to calculate coverage, and the cost of this overhead is incurred on each fault simulation run. Additional computation is also needed to compute the set intersection in every iteration, and both of these overheads will tend to scale with the number of coverpoints. So while a granular metric may expose more holes, it will also incur more overhead in simulation and post-processing.

D. Fault Model Considerations

The choice of fault model can affect both the runtime of the proposed approach as well as the quality of the result. In this subsection we will discuss both of these concerns together, as they are interrelated.

The primary determining factor of runtime for coverage discounting is the number of fault simulation runs, which is in turn determined by how many faults are simulated. The trivial solution is to simply not simulate all faults produced by a given fault model, but rather select a subset that will lead to an acceptable overall simulation time.

A better solution is to select a fault model which is more likely to produce interesting results. There are two key fault simulation results of interest:

1) We only gain information if the fault is not detected.
2) We only gain information if some coverpoints are no longer covered.

Addressing both of these objectives simultaneously can be problematic. As we noted in Sec. III-B, there exists a relationship where fault models that tend to produce hard-to-detect faults also have a tendency to produce redundant faults

(at least in the case of mutations). Redundant faults are in turn less likely to change which coverpoints are covered.

We can instead focus on only the second objective by choosing a fault model that is good at "uncovering" coverpoints. For a control-flow oriented implicit coverage metric, the mutation operators that change equalities might be a good choice since they tend to change control flow (hence the use of this pairing in the example of Sec. II).

For functional coverpoints, an ad-hoc fault model designed to specifically target individual coverpoints may prove more effective. Such a fault model would have an activation constraint similar to the trigger of the coverpoint, and a fault effect that preempts that coverpoint's triggering. This sort of ad-hoc fault model is used in the experimental results of Sec. V.

When runtime is not a concern, it may also be beneficial to use multiple fault models. The overall quality of the result (that is, the amount of coverage holes revealed) will depend on the fault model, but collecting them together into one large fault set could produce better overall results.

E. Simulation Environment

Because coverage discounting is an iterative process (like any other fault insertion-based method), the overall runtime is highly sensitive to the iteration count and time of each iteration. While the choice of fault model determines the iteration count, it can also be effectively reduced through parallelization (which is increasingly viable on today's multicore microprocessors). The other target for optimization is the fault simulation environment, which determines the time spent for each iteration.

One opportunity to speed up each iteration is to simulate at the highest level of abstraction available that is capable of capturing the coverage information. For example, a cycle-accurate architectural simulator will generally be faster than an RTL implementation, but will still contain enough resolution to calculate many implementation independent functional coverage metrics. This approach was used in the experimental results of Sec. V, where a cycle-accurate instruction set simulator is used in lieu of an RT- or gate-level description.

Another option for speeding up functional fault simulation is the use of dynamic slicing to determine whether a fault has been masked before a test terminates. Such an approach could also be used to perform a more fine-grained analysis of the coverage data, allowing for the elimination of individual coverpoint triggers. This would enable the discounted coverage method to work in more situations and discount frequently hit coverpoints, and is one possible future research direction.

V. Experimental Results

A. Experiment Setup

The OpenRISC CPU from opencores.org [20] was used as the experimental benchmark for these experiments due to the availability of a functional test set as well as a high-level instruction set simulator to more efficiently perform fault simulation. Two test sets were used in these experiments: the hand-written instruction set tests packaged with OpenRISC, and a set of random pseudofunctional test programs generated for the experiment.

The random pseudofunctional tests are random instruction streams of 300 instructions each. Twice during these streams, after 150 instructions and again at the end of the test, the contents of all general purpose registers are XORed together and output as a debug value with the `l.nop 3` trap. This is roughly same debug output mechanism as is used by the hand-written instruction set tests. The output debug values are compared with their expected values to determine whether the test passes. While more carefully directed random tests, such as those in [21], would better approximate a real verification environment, the development of such a test environment is beyond the scope of this paper.

The coverage metric used here is a set of unweighted functional coverpoints, one for each opcode. This metric was selected because testing each opcode is a basic test requirement for any microprocessor. Computation of this metric was implemented by instrumenting the instruction set simulator to count the number of times each opcode is executed. This instrumentation did not incur any measurable overhead.

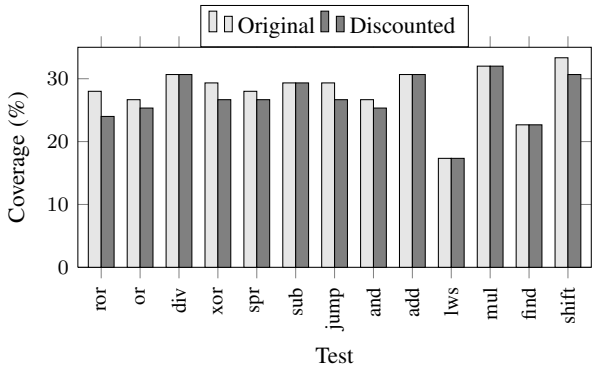
The fault model used is an ad-hoc model: it causes the DUT's instruction fetch to fail whenever a given opcode is encountered, such that the previous instruction will be repeated in its place. As we discuss in Sec. IV-D, this sort of targeted ad-hoc fault model allows the discounted coverage result to converge faster by preempting the triggering of specific coverpoints. It also limits the total fault population: only 734 faults were simulated, one targeting every opcode covered by each of 13 functional tests and 20 random pseudofunctional tests. Fault insertion was performed through additional instrumentation added to the instruction set simulator.

B. Test Quality: Hand Made vs. Random

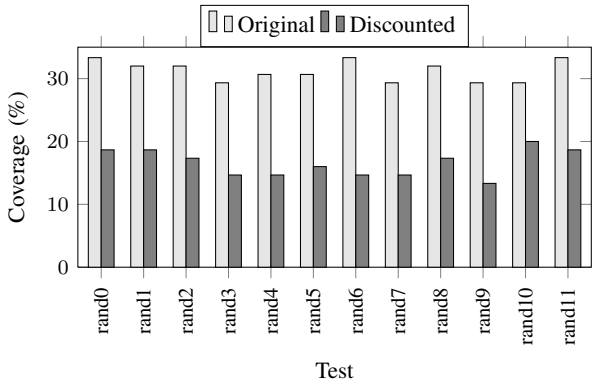
As in [9], we can evaluate our coverage methodology based on its ability to distinguish hand-made tests from random tests. This comparison accentuates the problems with activation-only coverage: random testing is very good at activating many functional corners, but hand-written tests tend to do a far better job of propagating the tested behavior to the output, where checkers can detect whether or not it is erroneous.

Fig. 1 shows the original and discounted functional coverage scores for each of the thirteen hand-written functional tests along with the first twelve randomly generated pseudofunctional tests. These results match with what we would anticipate: the functional test scores are not affected much by discounting, since they are written to propagate and thoroughly check the results of every operation. Note that the random tests start with a similar coverage score before discounting but, unlike the functional tests, lose about half of that score after discounting.

Table I summarizes both the per-test average scores (coverage before and after discounting) and the cumulative coverage for each test set, as well as the average and cumulative coverage for both test sets combined. It also lists the relative change in coverage score caused by discounting. Note that the



(a) Hand-Written Tests



(b) Pseudofunctional Tests

Fig. 1. Coverage of different test sets before and after discounting. Both test sets are approximately comparable before discounting. As expected, the hand-written functional tests (a) do a good job of propagating and checking the results. This can be seen in the relatively small change in coverage score after discounting. The random test programs (b) are not as carefully constructed, and coverage discounting exposes this shortcoming.

TABLE I
COVERAGE DISCOUNTING SUMMARY

Test Set	Functional		Random		Both (F+R)	
Value	Avg.	Total	Avg.	Total	Avg.	Total
Coverage (%)	28	57	31	93	29	100
Disc. Cov. (%)	27	57	17	51	20	69
Rel. Change (%)	4	0	47	46	31	31

total scores are not equal to the sum of the per-test coverage: any coverpoint covered by more than one test will only be counted once.

There are a number of details in Table I worth noting. First, observe that the while some functional tests experienced a small loss of coverage from discounting, the functional test set's total coverage did not change. This indicates that the discounted coverpoints from some tests were covered by others, as expected from a test set where each test is focusing on a specific function. Note also that the aggregate coverage of the random tests is higher than the functional test set before discounting but lower afterwards. While random tests are very good at exploring a wide range of functionality, there is no guarantee that they exercise it meaningfully.

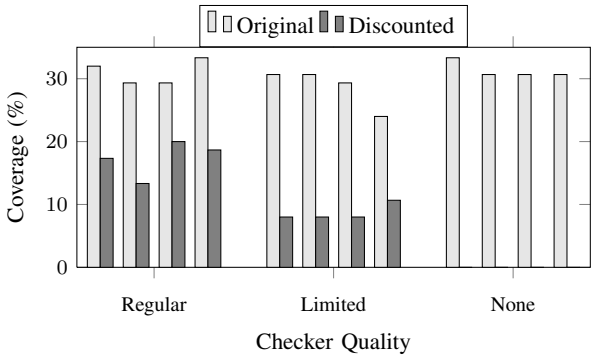


Fig. 2. Coverage for random tests with varying checker quality. The "regular" tests have a thorough checker which compares the entire output to its expected value. The "limited" tests have a checker which only compares the least significant bit of the output, and the "none" tests have no checker at all. Note that all 12 of these tests are different (since the checkers are partially integrated into the tests), leading to the variation in original coverage score.

C. Checker Quality

One of the benefits of coverage discounting over observability coverage metrics is the ability to diagnose checker deficiencies. To demonstrate this capability, additional random tests with less complete checkers but equal observability were generated. Fig. 2 shows the original and discounted coverage scores for 12 random pseudofunctional tests broken into three categories. The first category, "regular", has a checker that compares the entire output to its expected value. These tests are the same as rand8 - rand11 from Fig. 1b, and are repeated here to ease comparison with the other categories.

The second category, "limited" has a checker that compares the least significant bit of the output with its expected value. This leads to a decrease in coverage after discounting because a less thorough checker detects fewer faults, and therefore the tests as a whole have lower coverage. Finally, the "none" category contains tests without any output checker. This is the pathological case: it shows that nothing is meaningfully covered in the absence of checkers, and that coverage discounting correctly identifies this situation.

VI. CONCLUSION

In this paper, we described an analytical approach that combines activation-only coverage metrics with fault insertion techniques to produce a revised, or "discounted" coverage score that reflects the ability of the testbench to detect errors.

Discounted coverage effectively supersedes the next most related technique, observability coverage, in that it works with arbitrary functional coverage metrics, actually tests the quality of the testbench's checkers, and does not require complex instrumentation to compute. The runtime of our approach is on par with any fault insertion method such as mutation analysis, and can be sped up using similar approaches (such as parallelization or dynamic slicing).

Experimental results demonstrate the ability of the proposed technique to differentiate high quality tests from poor quality ones, even when the original coverage scores are similar. The

results also demonstrate the ability of coverage discounting to distinguish a good quality checker from a poor quality one, even when the overall observability is equal. Because the resulting coverage holes have a high-level functional meaning, they direct validation engineers more affectively than undetected faults, and without additional effort of redundant fault identification.

Future research directions include an analysis of different fault injection mechanisms, including commercially available fault insertion tools and the automated generation of ad-hoc fault models based on a set of functional coverpoints. We also hope to explore the use of dynamic slicing to speed up fault simulation and perform fine-grained discounting.

REFERENCES

[1] A. Gluska, "Coverage-oriented verification of banias," in *Proc. Design Automation Conference*. IEEE, 2003.

[2] R. Grinwald, E. Harel, M. Orgad, S. Ur, and A. Ziv, "User defined coverage-a tool supported methodology for design verification," in *Proc. Design Automation Conference*, 1998.

[3] R. DeMillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, 1978.

[4] B. Bailey, "Can mutation analysis help fix our broken coverage metrics?" *Cell*, 2008.

[5] M. Hampton and S. Petithomme, "Leveraging a commercial mutation analysis tool for research," in *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, 2007.

[6] P. Lisherness and K. T. Cheng, "Scemit: a systemc error and mutation injection tool," in *Proc. Design Automation Conference*. IEEE, 2010.

[7] A. Sen and M. Abadir, "Coverage metrics for verification of concurrent systemc designs using mutation testing," in *IEEE International High Level Design Validation and Test Workshop*, 2010.

[8] S. Tasiran and K. Keutzer, "Coverage metrics for functional validation of hardware designs," *IEEE Design & Test of Computers*, 2001.

[9] F. Fallah, S. Devadas, and K. Keutzer, "Occom-efficient computation of observability-based code coverage metrics for functional verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2001.

[10] T. Lv, J. Fan, X. Li, and L. Liu, "Observability statement coverage based on dynamic factored use-definition chains for functional verification," *Journal of Electronic Testing*, 2006.

[11] P. Lisherness and K. T. Cheng, "An instrumented observability coverage method for system validation," in *IEEE International High Level Design Validation and Test Workshop*, 2009.

[12] D. Van Campenhout, T. Mudge, and J. Hayes, "Evaluation of design error models for verification testing of microprocessors," in *IEEE International Workshop on Microprocessor Test and Verification*, 1998.

[13] A. Offutt and J. Hayes, "A semantic model of program faults," in *SIGSOFT Software Engineering Notes*. ACM, 1996.

[14] A. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1996.

[15] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, 2010.

[16] W. Howden, "Weak mutation testing and completeness of test sets," *IEEE Transactions on Software Engineering*, 1982.

[17] A. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," in *Proc. International Conference on Software Engineering*. ACM, 1993.

[18] A. Offutt and W. Craft, "Using compiler optimization techniques to detect equivalent mutants," *Software Testing, Verification and Reliability*, 1994.

[19] R. Hierons, M. Harman, and S. Danicic, "Using program slicing to assist in the detection of equivalent mutants," *Software Testing, Verification and Reliability*, 1999.

[20] "OpenRISC Project," http://opencores.org/project,or1k, 2011.

[21] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using bayesian networks," in *Proc. Design Automation Conference*. IEEE, 2003.