Cost-Effective Analysis of Post-Silicon Functional Coverage Events

Farimah Farahmandi[1], Ronny Morad[2], Avi Ziv[2], Ziv Nevo[2] and Prabhat Mishra[1]

[1]Computer and Information Science and Engineering
University of Florida, USA

[2] IBM Research
Haifa, Israel

***Abstract*—Post-silicon validation is a major challenge due to the combined effects of debug complexity and observability constraints. Assertions as well as a wide variety of checkers are used in pre-silicon stage to monitor certain functional scenarios. Pre-silicon checkers can be synthesized to coverage monitors in order to capture the coverage of certain events and improve the observability during post-silicon debug. Synthesizing thousands of coverage monitors can introduce unacceptable area and energy overhead. On the other hand, absence of coverage monitors would negatively impact post-silicon coverage analysis. In this paper, we propose a framework for cost-effective post-silicon coverage analysis by identifying hard-to-detect events coupled with trace-based coverage analysis. This paper makes three major contributions. We propose a method to utilize existing debug infrastructure to enable coverage analysis in the absence of synthesized coverage monitors. This analysis enables us to identify a small percentage of coverage monitors that need to be synthesized in order to provide a trade-off between observability and design overhead. To improve the observability further, we also present an observability-aware trace signal selection algorithm that gives priority to signals associated with important coverage monitors. Our experimental results demonstrate that an effective combination of coverage monitor selection and trace analysis can maintain the debugging observability with drastic reduction (up to 10 times) in the required coverage monitors.**

I. Introduction

The exponential growth of System-on-Chip (SoC) complexity, time-to-market reduction and huge gap between simulation speed and hardware emulation speed force the verification engineers to shorten the pre-silicon validation phase. There is a high chance that many bugs escape from pre-silicon analysis and it affects the functionality of the manufactured circuit. To ensure the correct operation of the design, post-silicon validation is necessary. However, post-silicon validation is a bottleneck due to limited observability, controllability and technologies to cope with future systems [1]. There is a critical need to develop efficient post-silicon validation techniques.

Assertions and associated checkers are widely used for design coverage analysis in pre-silicon validation to reduce debugging time. They can also be used in the form of coverage monitors to address controllability and observability issues in post-silicon. However, every coverage monitor introduces additional area, power and energy overhead that may violate the design constraints. To address these limitations, we propose a framework to reduce the number of coverage monitors in post-silicon while utilize the existing debug infrastructure to enable functional coverage analysis. To the best of our knowledge, our approach is the first attempt in utilizing debug infrastructure (trace data) for functional coverage analysis to enable a trade-off between coverage and overhead.

There are several built-in debug mechanisms such as trace buffers and performance monitors in order to enhance the design observability during post-silicon validation and reduce debugging efforts. Trace buffers record the values of a limited number of selected signals (typically less than 1% of all signals in the design) during silicon execution for specified number of clock cycles. The trace buffer values can be analyzed off-chip to restore the values of untraced signals. This paper makes three important contributions. First, we present an approach to utilize the information that can be extracted from on-chip trace buffer in order to determine easy-to-detect functional coverage events. Next, our trace-based coverage analysis enables the trade-off between observability and hardware overhead. We also propose a signal selection algorithm to improve the coverage analysis without compromising the observability of the whole design.

Although our proposed method can provide coverage data only for the recorded cycles (instead of the complete execution), it can significantly reduce the post-silicon validation effort for various reasons. First, the traced data for specific cycles are able to restore untraced signals for additional cycles. Moreover, when we know that certain events have been hit (covered) by trace analysis, the validation effort can be focused on the remaining set of events. From a practical perspective, it is not valuable to collect coverage data when generating testcase for an exerciser [8] or during checking a testcase since the exerciser code is relatively simple, repetitive and not expected to hit bugs on silicon. If the trace buffer is reconfigured to record signals during testcase execution, more insight about coverage events can be obtained. Our experimental results using ISCAS'89 benchmarks show that our approach can provide an order-of-magnitude reduction in design overhead without sacrificing functional coverage compared to when all assertions are synthesized.

The remainder of the paper is organized as follows. We discuss related work in Section II. Section III provides an overview about assertion-based validation. Section IV describes our post-silicon functional coverage analysis framework. Section V presents our experimental results. Finally, Section VI concludes the paper.

II. Related Work

Post-silicon validation techniques consider many important aspects such as effective use of hardware verification techniques [1] and stimuli generation [3]. Several approaches are also focused on test generation techniques [4], [5], [10] that can address various challenges associated with post-silicon debug. Pre-silicon assertions are converted (synthesized) to post-silicon coverage monitors by recent approaches [6], [7]. Coverage monitors can be reconfigured during run-time to change the focus of the observability. Unfortunately, synthesized coverage monitors can introduce unacceptable hardware overhead. Adir et at. proposed a method to utilize post-silicon exerciser on a pre-silicon acceleration platform in order to collect coverage information from pre-silicon [17]. However, the collected pre-silicon coverage may not accurately reflect post-silicon coverage in many scenarios.

978-3-9815370-8-6/17/$31.00

Knowledge of internal signal states during post-silicon execution helps to trace the failure propagation to debug the circuit. Trace buffers are used to sample a small set of internal signals since they can help in restoration of other signals and improve the design observability. There are different techniques to select trace signals such as structure/metric-based selection [11], [9], simulation-based selection, as well as hybrid of both approaches [12]. Recently, Ma et al. have proposed a metric that models behavioral coverage [13]. However, none of these approaches consider functional coverage analysis as a constraint for signal selection. Our proposed signal selection improves functional coverage analysis without compromising the debugging observability.

III. Background

Based on the functional coverage goal, a design is instrumented to check specific conditions of few internal signals. For example, assertions are inserted in a design to monitor any deviation from the specification. These days, designers mostly use one of the powerful assertion languages such as PSL (Property Specification Language) to describe interesting behavioral events (linear temporal logic assertions). Assertions can be classified into two groups: conditional and obligation [14]. The goal of a conditional assertions is to detect a failure. Therefore, it is activated every time all of its events are observed. For example, "$assert \quad never \quad b_1$" is called a conditional assertion as every time b_1 is evaluated true, a failure will happen and assertion should be triggered. On the other hand, an assertion is in obligation mode when a failure in its sequence triggers it. For example, "$assert \quad always \quad b_2$" is in obligation mode as b_2 should be always true and every time it is evaluated to false, the assertion is activated.

Example 1: Consider a part of a circuit shown in Figure 1. Suppose that we have two design properties: first, whenever signal F asserted, signal I is supposed to be asserted within next three cycles. The following assertion describes this property $A_1 : assert \quad always(F \rightarrow \{[*1 : 3]; I\}) \quad @rising\_edge(clk)$. Consider a second property where we would like to cover functional scenarios such that K and H signals are not true at the same time. This property can be formulated as $A_2 : assert \quad never(K\&H)$.

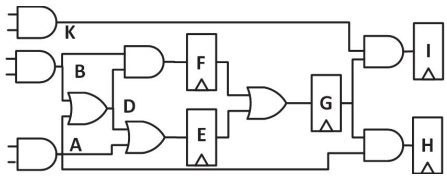


Fig. 1: A simple circuit to illustrate design properties

IV. Post-Silicon Functional Coverage Analysis

To have full observability in post-silicon, one option is to synthesize all of the functional scenarios (typically thousands of assertions, coverage events, etc.) to coverage monitors and track their status during post-silicon execution. However, this option is not practical due to unacceptable design overhead. Therefore, designers would like to remove all or some of the coverage monitors to meet area and energy budgets. It creates a fundamental challenge to decide which coverage monitors can be removed. We propose an approach to evaluate the assertion activation efforts by on-chip trace buffer and rank them based on the difficulty in covering/detecting them. Clearly, the hard-to-detect ones should be synthesized, whereas the easy-to-detect ones can be ignored (trace analysis can cover them).

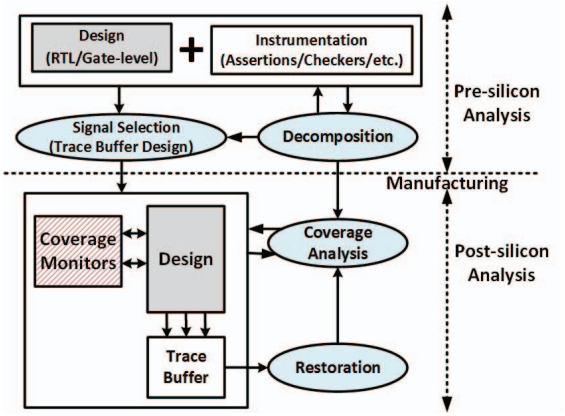


Fig. 2: Overview of our proposed approach.

Figure 2 shows an overview of our proposed method. Our proposed approach consists of four major steps: decomposition of coverage scenarios, signal restoration, coverage analysis and signal selection. The remainder of this section describes these steps.

A. Decomposition of Coverage Scenarios

Suppose that a gate-level design D as well as a set of pre-silicon RTL assertions $\mathbb{A}$ are given and our plan is to use trace buffer information to determine activation of $\mathbb{A}$ in model D during silicon execution. We propose an off-line functional event decomposition to enable post-silicon coverage analysis. The decomposition can be done in pre-silicon. First, RTL assertions are scanned to extract their signals and their corresponding gate-level signals based on name mapping methods. Next, each RTL assertion from set $\mathbb{A}$ is mapped to a set of clauses such that each clause contains assignments to a set of signals in specific cycles.

Formally, each pre-silicon assertion $A_i \in \mathbb{A}$ is scanned and its signals and its corresponding gate-level signals are defined. Then, A_i is decomposed to a set of clauses $A \equiv \mathbb{C} = \{C_1, C_2, ..., C_n\}$ based on its mode (conditional or obligation). Each C_j can be formalized as $C_j = \{\alpha_1\Delta_1\alpha_2\Delta_2...\Delta_{m-1}\alpha_m\}$. Each α_k presents a Boolean assignment on gate-level signal $n \subset \mathbb{N}$ ($\mathbb{N}$ shows all corresponding gate-level signals of set $\mathbb{A}$) on cycle c_t where $1 \leq c_t \leq CC$ (Suppose we know that the manufactured design will be simulated for maximum CC clock cycles) such as $\alpha_k : \{n = val \quad in \quad cycle[c_t]\}$ where $val \in \{0, 1\}$. Operators Δ_k can be one of the logical operations such as AND, OR or NOT. As a result, the original assertion is translated as a set of clauses $\mathbb{C}$ in a way that activation of one of them triggers the original assertion.

From now on, we assume that signals of A_i are mapped to corresponding assertion on gate-level signals. In order to generate each of C_j, Algorithm 1 is used. It partitions assertions based on their mode. If an assertion is in obligation mode, its original conditions are negated since we are looking for conditions that cause the assertion to fail (lines 7-8) and each C_j contains a subset of the negated conditions over different clock cycles that cause the assertions to fail. On the other hand, if the assertion is in conditional mode, the original assertions are kept as they are (lines 9-11). Each clause contains a subset of conditions which is logically equivalent to a set of conditions (lines 12-13). Clauses are also expanded over time and contain timing information (line 14). Therefore, each assertion is mapped to a set of clauses such that activation of one of the clauses leads to activation of

the original RTL assertion (line 15). Specifically, the following rules are used to generate the set of clauses ($\mathbb{C}$):

- If an assertion is in obligation mode and it contains an AND operator ($p \wedge q$), the operands are negated and AND will be changed to a OR operator ($p \vee q$). For example, in assertion "$assert \quad always \quad p \quad \& \quad q$", whenever conditions p or q are false, the assertion is activated. Therefore, the assertion is translated to a set of clauses as $\mathbb{C} = \bigcup_{t=1}^{CC}\{p = 0[t] \vee q = 0[t]\}$ (clauses are also expanded over time).
- If an assertion is in obligation and it contains OR operator such as "$assert \quad always \quad (p \quad | \quad q)$", the conditions will be negated and the set of clauses is extracted as: $\mathbb{C} = \bigcup_{t=1}^{CC}\{p = 0[t] \wedge q = 0[t]\}$.
- If an assertion is in obligation mode and it contains an implication operator, antecedent conditions are not modified. However, consequent conditions are negated. For example, if we have "$assert \quad always \quad (p \rightarrow next \quad q)$", it is converted to $\mathbb{C} = \bigcup_{t=1}^{CC}\{p = 1[t] \wedge q = 0[t+1]\}$. Next operation shows its effect in condition $q = 0[t+1]$.
- If an assertion is in conditional and it contains OR operator such as "$assert \quad never \quad (p \quad | \quad q)$", it is translated to $\mathbb{C} = \bigcup_{t=1}^{CC}\{p = 1[t] \vee q = 1[t]\}$.
- If an assertion is in conditional mode and it contains an AND operator ($p \wedge q$), the operation is kept as it is. For example, in assertion "$assert \quad never \quad (p \quad \& \quad q)$", if p and q are true at the same time, the assertion will be activated. Therefore, the assertion is translated as $\mathbb{C} = \bigcup_{t=1}^{CC}\{p = 1[t] \wedge q = 1[t]\}$.
- If an assertion is in conditional mode and it contains implication operator, antecedent and consequent conditions remain the same as the original assertion. For example, if we have "$assert \quad never \quad (p \rightarrow next \quad q)$", it is converted to $\mathbb{C} = \bigcup_{t=1}^{CC}\{p = 1[t] \wedge q = 1[t+1]\}$.
- If there are *eventually!* or *until* operators in an assertion, based on the mode of assertion it shows its effect in generating repeating conditions in different clock cycles. For example, "$assert \quad always \quad (p \rightarrow \quad eventually \quad q)$" is translated to $\mathbb{C} = \bigcup_{t=1}^{CC}\{(p = 1[t]) \wedge (q = 0[t] \wedge q = 0[t+1] \wedge ... \wedge q = 0[CC])\}$. On the other hand, if we have an assertion as "$assert \quad always \quad (p \quad until \quad q)$", the conditions are found as: $\mathbb{C} = \bigcup_{t=1}^{t+n=CC}\{(p = 1[t]) \wedge (p = 0[t+1] \vee p = 0[t+2] \vee ... \vee p = 0[t+n-1]) \wedge (q = 1[t+n])\}$.

Example 2: Consider the assertions in Example 1 form Section III. We assume that the circuit will be executed for 10 clock cycles for post-silicon validation. The first property (A_1) will be decomposed to equivalent conditions as follows: $\mathbb{C}_{A_1} : \{\{F = 1[1] \wedge I = 0[1] \wedge I = 0[2] \wedge I = 0[3]\}, \{F = 1[2] \wedge I = 0[3] \wedge I = 0[4] \wedge I = 0[5]\}, ..., \{F = 1[7] \wedge I = 0[7] \wedge I = 0[9] \wedge I = 0[10]\}\}$
The assertion is activated if signal *F* is asserted and signal *I* remains false for next three cycles. The second property is decomposed as shown below since it will be activated if both C and H are true at the same time.
$\mathbb{C}_{A_2} : \bigcup_{t=1}^{10}\{C = 1[t] \wedge H = 1[t]\}$

The computed conditions are used to detect activation of assertions during post-silicon validation as described in Section IV-C.

B. Restoration of Signal States

Suppose that we have a gate-level design with $\mathbb{G}$ internal signals and the design has been executed for CC clock cycles during post-silicon validation. A set of signals ($\mathbb{S} \quad where \quad \mathbb{S} \subset \mathbb{G}$) are sampled and their values are stored in trace buffer T during post-silicon execution for CC_t clock cycles ($CC_t \leq CC$). The information of the trace buffer (with $|\mathbb{S}|$ and CC_t dimensions) can be used to find the values of other signals ($\mathbb{G} - \mathbb{S}$). The restoration starts from the stored values of $\mathbb{S}$ signals over CC_t cycles and go forward and backward to fill the values of matrix $\mathbb{M}_{\mathbb{G}xCC}$. Matrix $\mathbb{M}$ is used to present states of the design during CC clock cycles. Each cell of matrix $\mathbb{M}$ can have value 0, 1 or X. Value X in $m_{i,j} \in \mathbb{M}$ presents the fact that the value of signal i in clock cycle j cannot be restored based on traced values of $\mathbb{S}$ sampled signals. We utilize matrix $\mathbb{M}$ information to determine if any of assertions is definitely covered during run-time.

Algorithm 1 Assertion decomposition algorithm

```
1:  procedure DecomposeAssertions
2:      Input: RTL assertions 𝔸
3:      Output: 𝒞 which maps each A_i ∈ 𝔸 to equivalent ℂ
4:      𝒞 = {}
5:      for each A_i ∈ 𝔸 do
6:          ℂ = {}
7:          if A_i is in obligation mode then
8:              Ω = FindFailureConditions(A_i, 𝔾)
9:          else
10:             /*A_i is in conditional mode*/
11:             Ω = FindPassingConditions(A_i, 𝔾)
12:         for every possible case do
13:             ℂ = ℂ∪ SubsetOfEquivalentConditions(Ω)
14:             addTiming(ℂ)
15:         𝒞.put(A_i, ℂ)
16: return 𝒞
```

C. Coverage Analysis

Our plan is to use both traced and restored values to check the clauses that we found in Section IV-A to define easy-to-detect assertions. In order to find coverage for assertions in set $\mathbb{A}$, we consider each assertion $A_i \in \mathbb{A}$ and find its corresponding decomposed clauses, set $\mathbb{C}$, as described in Section IV-A. Set $\mathbb{C}$ is designed in a way that if one of the $C_i \in \mathbb{C}$ can be evaluated to true on matrix $\mathbb{M}$, assertion A_i is triggered. Using the proposed method, each C_i contains a set of Boolean functions (α_j) and each $\alpha_j : n = val$ in cycle t where $1 \leq t \leq CC$ is mapped to one cell of matrix $\mathbb{M}$ ($m_{n,t}$). If the value of $m_{n,t}$ is equal to $val \in \{0, 1\}$, the condition α_j is evaluated true. Condition C_i is evaluated true when the expression consisting of all α_j and Δs evaluated to be true. An assertion is called covered during post-silicon validation if one of its C_is is evaluated to be true.

For assertions that originally contains implication operator ($A : assert \quad p \rightarrow q$), we keep the information that which Boolean α_j belongs to the precondition (p) and which conditions belongs to the fulfilling condition (q) when we want to check their conditions over $\mathbb{M}$. In order to check assertion A, we start from rows which belongs to signals existing in antecedent and check every cycle to find the desired value. Then, we continue the search for consequent from those cycles when antecedent is true to find values that makes whole A true. In other words, to be able to find out the activation of assertion A, we need to minimize the number of X values in cells of matrix $\mathbb{M}$. We also count how many times assertion A is activated for sure. Note that for checking conditions, 3-valued (ternary) logic is used. In other words, condition $p \vee q$ is evaluated as true if signal p is true and q has X value and vice

TABLE I: Restored signals for circuit shown in Figure 1 when A and B are trace signals.

Signal/Cycle	1	2	3	4	5	6	7	8	9	10
A	0	1	0	1	1	0	1	0	1	0
B	1	0	0	1	0	0	0	1	1	1
K	X	X	X	X	X	X	X	X	X	X
D	1	1	0	1	1	0	1	1	0	1
E	X	1	1	0	1	1	0	1	1	0
F	X	0	1	0	1	1	0	1	0	0
G	X	X	1	1	0	1	1	0	1	1
I	X	X	X	0	1	0	0	0	0	0
H	X	X	X	X	X	0	X	X	0	X

versa. Algorithm 2 presents the coverage analysis procedure. It counts the number of times an assertion is activated during execution. If *activationCount* is equal to zero, it means that we cannot determine activation of the assertion based on trace buffer values. The coverage (cov) percentage is computed by counting the assertions that have been activated at least once and dividing it by the number of total assertions.

Algorithm 2 Assertion coverage measurement algorithm

```
1: procedure CoverageAnalysis
2:     Input: Trace buffer T, trace signals 𝕊, gate-level signals 𝔾, condition map 𝒞, assertions 𝔸, max cycle CC
3:     Output: Coverage map Θ)
4:     T=Restoration(𝕊);
5:     𝕄 = ConstructDesignStatesMatrix(T, 𝔾, CC)
6:     for each A_i ∈ 𝔸 do
7:         activationCount = 0
8:         ℂ =𝒞.get(A_i)
9:         for each C_j ∈ ℂ do
10:            if checkCondition(C_j, 𝕄) then
11:                activationCount++
12:        Θ.put(A_i, activationCount)
13: return Θ
```

Example 3: Consider the circuit shown in Figure 1 and the associated assertions shown in Example 1. Suppose that the only two signals (A and B) can be traced during post-silicon validation (the width of trace buffer is two). Note that in signal selection part we are not limited to flip-flops and every internal signal can be considered as potential sampled signal. Table I shows the states of the design based on the stored values of A and B signals. In fact, Table I shows matrix $\mathbb{M}$. The restoration ratio is equal to: $65/(9*10) = 0.72$. In other words, 72.22% of internal states are restored. Suppose that we take the clauses shown in Example 2 and the matrix shown in Table I as inputs to compute the coverage of these assertions during run-time. Based on information shown in Table I, assertion A_1 is activated since signal F is asserted in cycle 5 and signal I remains zero in the next three cycles, 6, 7 and 8 (activation of A_1 is detected twice). However, we cannot comment on A_2 since the respective conditions cannot be evaluated.

Until now, we identified which assertions are activated during run-time for sure. We rank assertions based on the required efforts to detect them using our proposed method of Section IV-C to decide which assertions are better to be kept as coverage monitors in post-silicon to improve the design observability and increase the assertion coverage. The assertions that are hard-to-detect (for example, cannot be detected even one time using the proposed method) or represent critical functional scenarios are best candidates to be kept as coverage monitors in silicon to improve the design observability. Algorithm 3 presents the proposed approach. The algorithm sorts set $\mathbb{A}$ based on their activation count that obtained from Algorithm IV-C and priority (critical scenarios or assertions that their activation cannot be detected using trace buffer values have higher priority). Next, we select assertions that fits in area and power budgets and increase total coverage and add them to cov_mon. For example, if two assertions have same priority, we choose the one that has less number of operators and signals (represents less area overhead). The algorithm returns the set cov_mon as selected coverage monitors.

Algorithm 3 Coverage monitor selection algorithm

```
1: procedure Coverage monitor selection
2:     Input: Assertions 𝔸, desired coverage des_cov, budget for coverage monitors budg, gate-level signals 𝔾, trace buffer T with trace signal 𝕊, trace buffer width W
3:     Output: Selected coverage monitors cov_mon
4:     cov_mon = {}
5:     𝒞=DecomposeAssertions(𝔸)
6:     Θ=CoverageAnalysis(T, 𝕊, 𝔾, 𝒞, 𝔸, CC)
7:     cov=findCoverageNumber(Θ,𝔸)
8:     𝕌=findUndetectedAssertions(Θ,𝔸)
9:     SortBasedOnDetectionEfforts(𝔸,Θ)
10:    tmp_cost = cost, tmp_cov = cov
11:    while cov_t < des_cov && cost_t < budg do
12:        find a_i ∈ 𝔸 where a_i.selected = false and budg − cost_t − a_i.cost ≥ 0
13:        cost_t+ = a_i.cost
14:        a_i.selected = true
15:        if a_i ∈ 𝕌 then
16:            cov_t++
17:        cov_mon = cov_mon ∪ u_i
18: return cov_mon
```

D. Coverage-aware Signal Selection

Traditional signal selection methods select signals that have priority over other design signals as they may have a better restorability and more internal signals might be restored during the off-chip analysis. However, if we select trace signals that have better restorability on signals appear on assertions, we can increase the chance of finding the activation of assertions. We propose a signal selection algorithm emphasizes restorability of assertion signals to be able to improve the assertion-based coverage analysis. We show that this way of trace signal selection has a better impact of analysis of assertion coverage while it has a negligible effect on observability of the whole design.

Algorithm 4 Assertion-aware trace signal selection algorithm

```
1: procedure Signal-Selection
2:     Input: Assertions 𝔸, trace buffer width W, gate-level signals 𝔾
3:     Output: Selected Trace Signals 𝕊
4:     ℕ=findSignalsExistInAssertions(𝔸)
5:     Ψ = findNumberOfOccuranceOfEachSignal(ℕ)
6:     𝕊 = {}
7:     while 𝕊.size() < W do
8:         generate random tests I
9:         for each g_i ⊂ 𝔾 which is not in 𝕊 do
10:            calculate restoration of g_i ∪ 𝕊
11:        select n_i with maximum restorability on ℕ.
12:        𝕊 = 𝕊 ∪ n_i
13: return 𝕊
```

Algorithm 4 shows our proposed signal selection algorithms that improves assertion coverage analysis. In order to select

trace signals that have a better restorability on assertion signals, pre-silicon assertions $\mathbb{A}$ are scanned to find their signals (set $\mathbb{N}$) and their importance based on how many times a specific signal is repeated in map Ψ (lines 4 and 5). We modify existing simulation-based trace signal selection algorithm to select signals which has maximum restoration ratio on assertion signals ($\mathbb{N}$) based on the simulated values of random test vectors for several cycles. If there is a tie, we use the signal that has a higher value in set Ψ. The algorithm continues until it selects as many as W trace signals (lines 6-12). Please note that our approach (providing emphasis on assertion signals) can be applied on top of other signal selection algorithms as well.

Example 4: As it can be seen from Example 3, the activation status of assertion A_2 cannot be detected based on information of Table I. Using Algorithm 3, only K and F are selected as trace signals based on their good restorability on assertion signals ($K, F, I \quad and \quad H$). Restoration and coverage analysis (using the traced values of new signals) would be able to detect activation of both assertions in Example 1.

V. Experiments

A. Experimental Setup

In order to evaluate the efficiency of our proposed approach, we have implemented our assertion decomposition, restoration, coverage analysis and signal selection algorithms using C++. We have applied our proposed methods on ISCAS'89 benchmarks (since most of the existing signal selection algorithms work with only these benchmarks). The trace buffers were chosen with a widths of 8, 16 and 32 and depth of 1024[1]. We have generated assertions both in obligation and conditional modes based on the presented method of [15]. Assertions were decomposed as a set of clauses using Algorithm 1. We simulated the benchmarks using different trace buffers for 1024 clock cycles with random test vector to model post-silicon validation. Trace signals were chosen based on our implementation of the presented method in [16] since it is the most recent signal selection approach. In the next step, we dumped the stored values of trace buffer and we tried to restore the values of unsampled signals over different clock cycles to construct the matrix representing the states of the circuit. Next, the assertion conditions were checked over matrix and we counted the number of activated assertions during run-time based on Algorithm 2. Finally, we used the signals selected by Algorithm 3 to further improve functional coverage analysis. Moreover, we use Algorithm 3 to selectively synthesize some coverage monitors which are more beneficial for improving functional coverage.

B. Results

Table II presents results for assertion coverage of total 12000 (4000 for each trace buffer configuration) assertions for each benchmark. The first three columns show the type of the benchmark, the number of its gates and the width of its trace buffer, respectively. The fourth column shows the restoration ratio based on existing trace signal selection. The fifth column shows the coverage of assertions using trace buffer (without introducing any overhead). Note that, *Observability-aware SS* represents our assertion coverage analysis framework on top of existing signal selection techniques. We improved the assertion coverage using our proposed signal selection algorithm with negligible effect on restorability of whole design (sixth and seventh columns). Note that, *Coverage-aware SS* represents our assertion coverage analysis framework on top of our coverage-aware signal selection method. Since signal selection algorithms are based on heuristic methods, in some of the cases, our coverage-aware signal selection algorithms improves the restorability of the design (such as s15850).

If we zoom in on each row of Table II, the activation details of each type of assertions are reported in Table III. The first three columns of Table III show the type of the benchmark, the number of its gates and the width of its trace buffer, respectively. The fourth and ninth columns show the restoration of the design using existing trace signal selection method as well as our proposed signal selection algorithms, respectively. The fifth and tenth columns show the coverage of one thousand single variable assertions (consisting of only one condition) on specific clock cycles. The sixth and eleventh columns show the coverage of one thousand two-variable assertions with AND operators where their conditions have templates of $\{n_1 = val_i \quad [c_i] \& \quad n_2 = val_j \quad [c_j]\}$ using both signal selection methods. The seventh and twelfth columns show the coverage of one thousand three-variable assertions with AND operators where their conditions are in the form of $\{n_1 = val_i \quad [c_i] \& \quad n_2 = val_j \quad [c_j] \& \quad n_3 = val_t \quad [c_t]\}$ respectively. The eighth and thirteenth columns show the coverage of one thousand three-variable assertions with OR operators where their conditions are in the form of $\{n_1 = val_i \quad [c_i] | \quad n_2 = val_j \quad [c_j] | \quad n_3 = val_t \quad [c_t]\}$ respectively. The results demonstrated the fact that using our approach enables designers to achieve significant functional coverage (up to 93%, 58% on average) without synthesizing any coverage monitors.

TABLE II: Assertion coverage when the total number of assertions for each row is 4000 (12,000 per benchmark)

Benchmark			Signal Selection			
			Observability-aware SS		Coverage-aware SS	
Type	#gates	#Traces	Restoration%	Asser. Cov.%	Restoration%	Asser. Cov.%
S5378	2995	8	60.97	46.97	58.57	49.6
		16	79.27	63.6	76.95	64.23
		32	93.10	88.5	92.26	90.57
S9234	5844	8	84.85	65.4	76.03	65.8
		16	90.19	75.27	83.70	83.12
		32	94.54	90.67	93.8	93.45
s15850	10383	8	72.03	59.7	76.03	65.8
		16	80.97	61.6	75.55	68.7
		32	84.14	72.9	82.66	74.9
s35932	17828	8	41.09	26.825	41.63	27.02
		16	41.35	26.825	41.88	27.05
		32	41.79	26.825	42.22	27.25
s38417	23843	8	36.53	23.025	36.97	23.23
		16	43.76	28.575	46.91	32.58
		32	49.77	35.075	55.82	42.33
s38584	20717	8	72.97	47.625	67.78	59.8
		16	79.15	63.65	76.53	69.3
		32	88.85	73.65	87.27	82.78

C. Observability versus Hardware Overhead

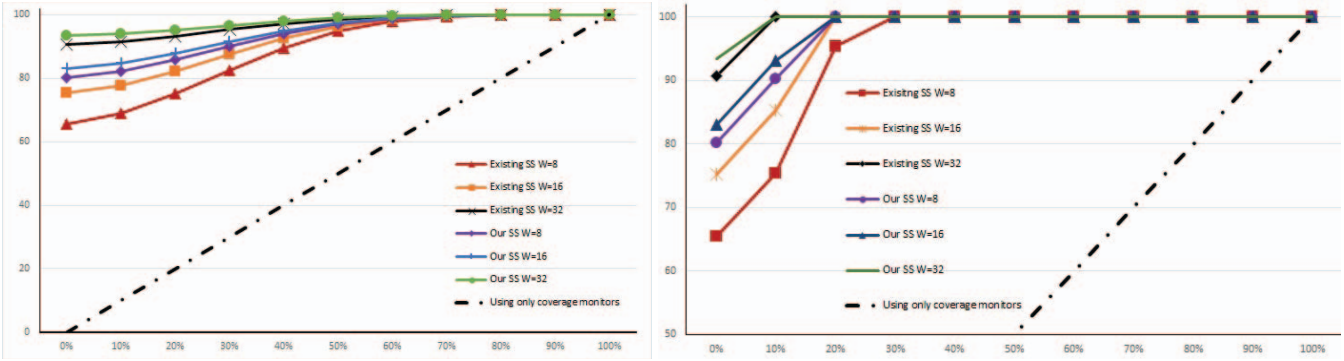
The results shown in Tables II and III show the extent of functional coverage analysis without introducing any hardware penalty for synthesized coverage monitors. Figure 3(a) shows coverage improvement if we randomly choose 10% to 90% of the remaining assertions that we cannot be sure about their activation using trace buffer information. The straight line shows the coverage when our method is not used and observability is provided only by using synthesized coverage monitors (the percentage of observability is equal to the percentage of synthesized assertions). On the other hand, we used Algorithm 3 to select hard to detect coverage monitors.

As it can be seen in Figure 3, 100% observability is achieved with significant reduction in overhead (40-50% coverage monitors with observability-aware signal selection can provide 100% functional coverage). Figure 3(b) shows the result of observability of s9234 with different trace buffer widths and different coverage monitor selection strategies (the straight line is cut on 50% for improved illustration). As it can be seen, when our signal selection algorithm was used to

[1]A trace buffer with width 32 and depth 1024 represents that it can trace the values of 32 signals over 1024 clock cycles.

TABLE III: Assertion coverage using trace buffer information

			Observability-aware SS					Coverage-aware SS				
Benchmark	#gates	#Trace	% restored Gates	% Single Variable Assertion	% Two Variable Assertion (AND)	% Three Variable Assertion (AND)	% Three Variable Assertion (OR)	% restored Gates	% Single Variable Assertion	% Two Variable Assertion (AND)	% Three Variable Assertion (AND)	% Three Variable Assertion (OR)
S5378	2995	8	60.97	63.6	36.2	25.4	62.7	58.57	64.2	42.6	25.3	66.3
		16	79.27	77.5	56.3	41.9	78.7	76.95	79.6	58.8	44	74.5
		32	93.10	92.6	86.5	80.9	94.0	92.26	94.5	88.4	84.8	94.6
s9234	5844	8	84.85	77.9	59.7	45	79	76.03	88.0	77.4	68.1	87.5
		16	90.19	85.5	71.1	58.9	85.6	83.7	88.6	81.4	72.6	89.9
		32	94.54	93.5	90.3	84.8	94.1	93.8	94.8	90.0	84.5	93.7
s15850	10383	8	72.03	73.9	54.8	37.1	73.0	76.03	76.5	61.7	46.9	78.1
		16	80.97	75.1	55.5	40.1	75.7	75.55	81.1	63.1	50.2	80.7
		32	84.14	82.2	68.1	56.6	84.6	82.66	84.2	70.6	59.5	85.3
s35932	17828	8	41.09	41.3	16.2	7.7	42.1	41.63	40.9	17.3	7.4	42.5
		16	41.35	41.3	16.2	7.6	42.2	41.88	40.9	17.3	7.4	42.5
		32	41.79	41.3	16.2	7.7	42.1	42.22	41.3	17.4	7.5	42.8
s38417	23843	8	36.53	38.2	13.8	3.9	36.2	36.97	37.7	13.9	5.8	35.9
		16	43.76	44.5	19.7	8.1	42	46.91	46.91	24.2	10.5	48.7
		32	49.77	51.2	26.5	12.4	50.2	55.82	56.3	35.5	19.2	58.3
s38584	20717	8	72.97	60.7	41.2	24.1	72.97	67.78	70.8	54.3	37.7	76.4
		16	79.15	75.3	59	43.4	76.9	79.69	79.7	65.3	51.2	81
		32	88.85	83.5	68.5	56.7	85.9	87.27	90.0	79.5	71.7	89.9



(a) Coverage monitors are selected randomly

(b) Coverage monitors are selected from hard-to-detect events

Fig. 3: Coverage analysis for s9234

choose 32 trace signals and our coverage monitor algorithm was used, synthesizing only 10% assertion leads to 100% observability. Although, we presented the result for s9234, we obtained similar results for other ISCAS89 benchmarks.

It can be argued that it takes little effort to cover the first 90%, but significantly more to cover the remaining 10%. Based on our proposed method, if the remaining 10% of the assertions are synthesized, 100% coverage is achieved. However, if it is not possible to synthesize those assertions due to design constraints, increasing the width or depth of trace signals can be considered. Dynamic signal selection capability (if available) can be utilized to focus in tracing of the remaining 10% assertions.

The experimental results demonstrated three important aspects of our approach. We provided a technique to improve the design observability when designers have a limited budget for synthesized coverage monitors. We showed that if they synthesize hard-to-detect assertions, the observability improves significantly. Our assertion-aware signal selection algorithm improves the assertion-coverage compared to existing signal selection techniques.

VI. Conclusion

We presented an approach to efficiently find functional coverage on silicon without introducing any overhead. The proposed method utilizes the existing debug infrastructure in modern designs to rank coverage monitors in terms of required efforts to detect them. We proposed a framework for trace-based functional coverage analysis. We explored the trade-off between observability and design overhead of synthesized coverage monitors. We also introduced a signal selection algorithm to improve the coverage analysis with negligible impact on restoration ratio. Our experimental results demonstrated that efficient ordering and selection of coverage monitors can drastically reduce (up to 10 times) design overhead without sacrificing functional coverage.

VII. Acknowledgments

This work was partially supported by the NSF grants (CCF-1218629 and CNS-1441667), SRC grant (2014-TS-2554), and an IBM Faculty Award.

References

[1] S. Mitra et al., "Post-silicon validation opportunities, challenges and recent advances," in *DAC*, 2010.
[2] A. Adir et al., "Threadmill: A post-silicon exerciser for multi-threaded processors," in *DAC*, 2011.
[3] A. Adir et al., "A unified methodology for pre-silicon verification and post-silicon validation," in *DATE*, 2011.
[4] M. Chen et al., *System-level Validation - high-level modeling and directed test generation techniques*, Springer, 2012.
[5] F. Farahmandi et al., "Exploiting transaction level models for observability-aware post-silicon test generation," in *DATE*, 2016.
[6] K. Balston et al., "Post-silicon code coverage for multiprocessor system-on-chip designs," in *IEEE Transactions on Computers*, 2013.
[7] M. Boule et al., "Adding debug enhancements to assertion checkers for hardware emulation and silicon debug," in *ICCD*, 2006.
[8] W. Kadry et al., "Comparative study of test generation methods for simulation accelerators," in *DATE*, 2015.
[9] K. Rahmani et al., "Efficient Selection of Trace and Scan Signals for Post-Silicon Debug," IEEE Trans. on VLSI, 2016.
[10] F. Farahmandi and P. Mishra, "Automated Test Generation for Debugging Arithmetic Circuits," DATE, 2016.
[11] K. Basu and P. Mishra, "RATS: Restoration-aware trace signal selection for post-silicon validation," in *VLSI Systems*, 2013.
[12] M. Li and A. Davoodi, "A hybrid approach for fast and accurate trace signal selection for post-silicon debug," in *DATE*, 2013.
[13] S. Ma et al., "Can't see the forest for the trees: State restoration's limitations in post-silicon trace signal selection," in *ICCAD*, 2015.
[14] M. Boulè and Z. Zilic, "Automata-based assertion-checker synthesis of psl properties," in *ACM Transaction Design Autom. Electr. Syst.*, 2008.
[15] L. Liu et al., "Towards coverage closure: Using goldmine assertions for generating design validation stimulus," in *DATE*, 2011.
[16] S. BeigMohammadi et al., "Combinational trace signal selection with improved state restoration for post-silicon debug," in *DATE*, 2016.
[17] A. Adir et al., "Reaching Coverage Closure in Post-Silicon Validation," in *HVC*, 2010.