Assertion-Based Validation using Clustering and Dynamic Refinement of Hardware Checkers

SAHAN SANJAYA, University of Florida, USA
HASINI WITHARANA, University of Florida, USA
PRABHAT MISHRA, University of Florida, USA

Post-silicon validation is a vital step in System-on-Chip (SoC) design cycle. A major challenge in post-silicon validation is the limited observability of internal signal states using trace buffers. Hardware assertions are promising to improve observability during post-silicon debug. Unfortunately, we cannot synthesize thousands (or millions) of pre-silicon assertions as hardware checkers (coverage monitors) due to hardware overhead constraints. Therefore, we need to select the most profitable assertions based on design constraints. However, the design constraints can also change dynamically during the device lifetime due to changes in use-case scenarios as well as input variations. Therefore, assertion selection needs to dynamically adapt based on changing circumstances. In this paper, we propose an assertion-based post-silicon validation framework to address the above challenges. Specifically, this paper makes two important contributions. We propose a clustering-based assertion selection technique that can select the most profitable pre-silicon assertions to maximize the fault coverage. We also present a cost-aware dynamic refinement technique that can select beneficial hardware checkers during runtime based on changing design constraints. Experimental evaluation demonstrates that our proposed pre-silicon assertion selection can outperform state-of-the-art assertion ranking methods (Goldmine and HARM). The results also highlight that our proposed post-silicon dynamic refinement can accurately predict area (less than 5% error) and power consumption (less than 3% error) of hardware checkers at runtime. This accurate prediction enables the identification of the most profitable hardware checkers based on design constraints.

1 INTRODUCTION

Post-silicon validation is widely used to detect and fix bugs in integrated circuits after manufacturing. Due to the increasing design complexity, it is infeasible to detect all functional as well as electrical bugs during pre-silicon validation [2]. Therefore, post-silicon validation is an essential step in SoC design methodology. One of the biggest challenges in post-silicon validation is the limited observability of internal states. Typically, a small trace buffer is used to trace a few hundred signals (out of millions of signals) during runtime [3, 4]. A prominent avenue to improve post-silicon observability is to use hardware checkers (assertions). According to the 2020 Wilson research study [5], around 75% of ASIC design and 50% of FPGA design projects use assertion-based validation [6, 7]. However, assertions also introduce hardware overhead. Therefore, it is not practical to synthesize thousands or millions of pre-silicon assertions to post-silicon checkers.

A promising avenue is to select a small set of pre-silicon assertions that can maximize the number of covered unexpected or invalid behaviors (non-vacuous failing states) during post-silicon debug. In other words, a higher number of covered non-vacuous failing states indicates a higher fault coverage. It is important to note that the selected assertions may not remain equally beneficial since the design constraints can change dynamically during the device's lifetime due to changes in use-case scenarios as well as input variations. Therefore, any static selection mechanism will not meet the objectives under changing design constraints. For example, mobile phone usage patterns can drastically change between two users. Even for the same user, the usage of the phone varies during different time periods of the day. In other words, different use-case scenarios and input variations can lead to dynamic changes in power and performance. Moreover, the dynamic changes

This is an extended version of the paper that appeared in DATE 2023 [1].
Authors' addresses: Sahan Sanjaya, University of Florida, Gainesville, FL, 32611, USA, ssanjaya@ufl.edu; Hasini Witharana, University of Florida, Gainesville, FL, 32611, USA; Prabhat Mishra, University of Florida, Gainesville, FL, 32611, USA.

in design constraints can limit the resources available for the hardware checkers. For example, the checkers can be disabled when the phone battery is low, which compromises the run-time checking capability. If the reconfigurability is available [8], it would be beneficial to dynamically refine the checkers to satisfy both dynamically changing circumstances and runtime checking objectives.

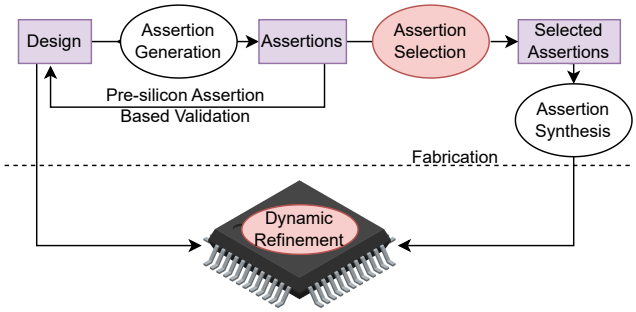


Fig. 1. Assertion-based post-silicon validation framework. The colored ovals (assertion selection and dynamic refinement) show our proposed contributions.

Figure 1 shows an overview of our proposed assertion-based post-silicon validation framework. During the pre-silicon stage, assertions can be generated automatically [6, 7, 9, 10] or manually for a given design. We propose a clustering-based assertion selection to minimize the set of assertions while maintaining considerable fault coverage. The selected assertions can be synthesized as hardware checkers. The hardware checkers are used as inputs for dynamic refinement. It may be infeasible to compute the design overhead of all possible combinations of hardware checkers. Instead, we train a regression model to predict the cost (power and area) of synthesizing a set of checkers. When the design constraints change dynamically, we use the regression model to select the most profitable subset of checkers that satisfies the design constraints at that time. The selected checkers are synthesized in reconfigurable hardware. Specifically, this paper makes the following major contributions:

- We propose a clustering-based selection of pre-silicon assertions to minimize the set of assertions while maintaining considerable fault coverage. Specifically, we first extract beneficial features to cluster similar assertions. Next, we construct the final set of assertions by selecting the dominating assertion from each cluster.
- We formulate the dynamic refinement problem as a cost-based non-linear optimization problem. Specifically, we utilize regression-based machine learning to perform cost prediction of post-silicon hardware checkers. We solve the non-linear optimization problem using gradient descent with simulated annealing.
- Extensive experimental evaluation demonstrates the suitability of pre-silicon assertion minimization and dynamic refinement of hardware checkers.

2 BACKGROUND AND RELATED WORK

We first provide relevant background on three automated assertion generation techniques with assertion ranking capability. Next, we survey existing assertion selection techniques as well as post-silicon assertion synthesis methods.

2.1 Assertion Generation using Goldmine

Goldmine [10, 11] is an automated assertion generation framework that utilizes static analysis and data mining techniques to automatically generate assertions. First, the design under test is simulated using random inputs to generate behavioral data. Next, design constraints such as cone-of-influence and topographical variable ordering are extracted through static analysis. The core assertion miner

employs decision tree-based supervised learning [12] to analyze the simulation traces and design constraints to produce candidate assertions. These candidate assertions are verified through a formal verification engine that supports SMV [13]. Finally, the verified assertions are ranked using support and confidence metrics.

2.2 Automatic Template-based Assertion Miner (A-TEAM)

A-Team introduces a template-based approach for assertion mining [14]. This method addresses issues such as limited flexibility caused by pre-defined templates and the presence of redundant assertions. First, the Apriori algorithm [15] is used to mine atomic propositions with high frequencies. Next, LTL assertions are mined by combining the atomic propositions obtained in the previous step. These combinations adhere to the user-provided templates. Finally, the mined assertions undergo evaluation against fault coverage, and assertions that do not improve fault coverage are discarded. A minimization process is applied to eliminate redundant assertions.

2.3 Hint-based Assertion Miner (HARM)

HARM [9] is a hint-based automated assertion generation framework. HARM generates LTL assertions based on a collection of user-defined hints and simulation traces of the design under verification. This tool does not need the actual source code of the design since the assertion generation relies on the traces. The user-defined hints consist of LTL templates, propositions, and ranking metrics, which are utilized by the assertion miner to reduce the search space and enhance the quality of the generated assertions.

2.4 Ranking of Pre-Silicon Assertions

Goldmine has implemented importance/complexity-based ranking (IRank) and statement coverage-based ranking (SRank) methods. SRank [16] is the ratio between the number of statements of a Verilog program covered by an assertion and the total number of statements in the design, including blocking, non-blocking, and conditional. IRank [16] is the ratio between the assertion importance with respect to a target variable and the assertion complexity. Goldmine ranks assertions with respect to a target variable. Therefore, it is challenging to compare two or more sets of assertions with different target variables to select the best minimized set of assertions. Moreover, Goldmine only supports a limited number of assertion templates and does not support variables with non-Boolean data types.

A-TEAM [14] has the ability to select the minimum number of assertions to maximize fault coverage. The primary strategy behind this minimization is to avoid assertions whose fault coverage can be claimed by others. One major drawback of this method is that it requires simulation of the design with testable faults to find the fault coverage. Also, A-TEAM does not provide individually ranked assertions. Therefore, it is hard to find a small set of assertions that can maximize the fault coverage from the resultant assertions.

HARM [9] supports a configurable and general context-based approach to rank the generated assertions. Users can define ranking and filtering metrics using different built-in assertion features. HARM allows multiple ranking metrics in a single ranking process, enabling the selection of assertions that perform well for most of the ranking metrics. The authors in [17] have introduced a new metric to rank assertions based on the quality of the assertions [18], which indicates the fault coverage. The main challenge in this minimization method is the long simulation time because it requires injecting some faults into the design and simulation to find the fault coverage of each assertion.

IMMizer [19] has the capability of minimizing the number of assertions by analyzing propositions in the given assertion set and extracting a new assertion set based on identified contradictory terms. Our method prioritizes post-silicon debug perspective, valuing each assertion for its contribution

to debugability and controllability. Minimizing the number of hardware checkers by combining some of them has the potential to reduce overall debugability. Therefore, in our proposed method, we aim to identify fault coverage redundancy among given manually or automatically generated assertions, which are later used as hardware checkers. The work in [20] proposes a promising fault coverage-based hardware checker selection method. Since the hardware checkers ranking method is based on bit-flip coverage, their method requires simulation or emulation results to find the most suitable hardware checkers. In contrast, our approach is based on static analysis, and therefore, it does not require any simulation or emulation results, which significantly reduces the assertion selection time. *Our proposed clustering-based assertion selection can minimize the number of assertions while maintaining considerable fault coverage without requiring simulation.*

2.5 Refinement of Post-Silicon Checkers

Pre-silicon assertions can be utilized during post-silicon debug by synthesizing them as hardware checkers [2]. A major challenge in assertion selection is to determine which assertions should be added to the design as hardware checkers. The number of hardware checkers can be reduced [21–23] by utilizing the existing debug infrastructure (trace buffer). Another promising alternative for cost-effective hardware checkers is the dynamic synthesis of checkers using FPGA [24]. In this work, the hardware checkers are included in a re-configurable embedded block (FPGA) [8] in a time-multiplexed manner. This approach enables the addition of a large number of checkers with a low area overhead. The authors in [25] convert formulas in LTL over finite traces into automata implementations on FPGAs for high-performance runtime monitoring. All of these approaches rely on static optimization results that may not be optimal under changing circumstances during runtime. *To the best of our knowledge, our approach is the first attempt to dynamically refine hardware checkers based on changing design constraints.*

3 PROBLEM FORMULATION

We first define few terms and concepts used in this paper. Next, we construct the problem formulation for both assertion selection and dynamic refinement.

3.1 Preliminaries

Definition 1: An *assertion* can be viewed as a function, *assert(expression)*, which fails when the *expression* evaluates to false. Examples of assertions include *assert(req)*, *assert(req → ack)*, etc. In this paper, we primarily focus on linear temporal logic assertion in the form of $assert$ (*antecedent* → *consequent*), since the existing assertion generators also produce the same type of assertions.

Definition 2: *Non-vacuous failing state* of an assertion is a Boolean state where *antecedent* becomes true and *consequent* becomes false. Mathematically, an implication is considered failed (false) even if the antecedent is false. However, in verification, there is no vacuous failing state. Non-vacuous failing state represents an unexpected or invalid behaviour in the given design. If an assertion has a large number of non-vacuous failing states, it is likely to capture more unexpected or invalid behaviors. Therefore, such an assertion is expected to provide higher fault coverage.

Definition 3: Both antecedent and consequent are some combinations of propositions. A *proposition* is a statement that is either true or false, but not both. Propositions can be combined using Boolean operators (such as &, |, !) and logical operators (such as implication and equivalence). For example, consider propositions P_1: $(illegal\_instr == 1)$, P_2: $(instr < 428)$, P_3: $(instr > 228)$, and P_4: $(!is\_compressed)$. They can be combined to form new propositions, such as "P_1&P_2", "$P_3 \rightarrow P_4$", "P_1 ##5 P_4", etc.

Definition 4: Any proposition of the form P_x ##N P_y can be mapped to P_x $[-N]$ & P_y, where P_x $[-N]$ indicates that we check the value of P_x N clock cycles earlier relative to P_y. Note that

now P_x and P_x $[-N]$ are considered as two different propositions. For example, the combined proposition (P_1 ##2 P_1 ##1 P_1) is mapped into (P_1 $[-3]$ & P_1 $[-1]$ & P_1). Here, P_1 $[-3]$, P_1 $[-1]$, and P_1 are considered as three different propositions.

Definition 5: A set of assertions can be defined as $A = \{a_1, \ldots, a_i, \ldots, a_o\}$. Here $o = n(A)$, where n(A) is the cardinality of the set A, and a_i is a linear temporal logic assertion in the form of *antecedent* $\rightarrow$ *consequent*.

Definition 6: We can construct m mutually exclusive subsets (clusters) $A_1, A_2, \ldots, A_h, \ldots, A_m$ based on the similarity of assertions in terms of covered non-vacuous failing states, where $1 \leq m \leq n(A)$, and $n(A_1) + n(A_2) + \ldots + n(A_m) = n(A)$. In other words, every element in a given subset exhibits full or partial redundancy in terms of covered non-vacuous failing states with other elements in the same subset.

Definition 7: Consider a subset (cluster) A_h with l similar assertions, $\{a_1, a_2, \ldots, a_l\}$. We select a_d as the *dominating assertion* in A_h if the set of covered non-vacuous failing states by assertions $a_1, a_2, \ldots, a_l$ excluding a_d, is a subset of covered non-vacuous failing states by a_d.

Definition 8: The set U represents the set of unique propositions that are used in assertions of set A. For example, if the set $A = \{P_1 \& P2 \rightarrow P_3, P_2 \rightarrow P_3 \& P_4,\}$, then $U = \{P_1, P_2, P_3, P_4\}$.

Definition 9: Let us define U_s^{at} and U_s^{ct} as the set of antecedent and consequent propositions, respectively, in the assertion a_s. Similarly, $P_{s,i}^{at}$ and $P_{s,i}^{ct}$ represent the i-th proposition in the set U_s^{at} and U_s^{ct}, respectively. Both U_s^{at} and U_s^{ct} are subsets of U. For example, if we have an assertion $a_9 = (P_1 \& P_2 \& P_3) \rightarrow (P_4 \& P_5)$, $U_9^{at} = \{P_1, P_2, P_3\}$ and $U_9^{ct} = \{P_4, P_5\}$.

Definition 10: The set B_s^{at} is defined as the set of Boolean states, where the antecedent of the assertion a_s becomes true. Similarly, B_s^{ct} represents the set of Boolean states, where the consequent of the assertion a_s becomes false. Elements of sets B_s^{at} and B_s^{ct} are represented as binary values with $n(U)$ number of bits, where each bit represents the Boolean value of a proposition in U. For example, if the assertion $a_{10} = P_1 \rightarrow P_2$, and $U = \{P_1, P_2, P_3\}$, then $B_{10}^{at} = \{100, 101, 110, 111\}$, and $B_{10}^{ct} = \{000, 001, 100, 101\}$.

Definition 11: The non-vacuous failing states of the assertion a_s can be expressed as the set $B_s^{at} \cap B_s^{ct}$. For example, the set of non-vacuous failing states of the assertion a_{10}, as defined in Definition 10, is the set $B_{10}^{at} \cap B_{10}^{ct} = \{100, 101\}$.

3.2 Problem Formulation: Assertion Selection

The objective of assertion selection is to *select the minimum number of assertions while maintaining considerable fault coverage*. The goal is to select m assertions from the set A that has o assertions, where $m << o$. There are $\frac{o!}{m!(o-m)!}$ possible combinations to select the m assertions. This number of combinations increases significantly as the size of the initial assertion set A grows. If we select m assertions randomly, there is no guarantee that the selected assertions will meet our objective. Therefore, it is critical to rank the assertions based on their ability to cover faulty behaviors and select the top-ranked assertions. As highlighted in Section 2, there are major challenges in state-of-the-art assertion ranking schemes. In Section 4, we propose a framework to first construct m subsets (clusters), where each cluster includes similar assertions as outlined in Definition 6. The final minimized set of assertions consists of the collection of all the dominating assertions selected from each subset as outlined in Definition 7. Specifically, we need to solve:

$$\underset{1 \leq i \leq m}{constructSubset}(\mathcal{A}_i)$$

$$\underset{1 \leq i \leq m}{selectDominator}(\mathcal{A}_i)$$

Two major challenges exist in solving the assertion selection problem: (i) how to cluster a set of similar assertions without simulation, and (ii) how to find the dominating assertion in each cluster? We propose solutions to address these challenges in Section 4.1, Section 4.2, and Section 4.3. The selected assertions are then synthesized as hardware checkers. Note that the design constraints may change dynamically during the device's lifetime, and the synthesized assertions should be able to adapt to these changes as outlined next.

3.3 Problem Formulation: Dynamic Refinement

Cost-based optimization is a powerful technique to address the problem of selecting a set of choices. It consists of associating costs with various choices and then finding the subset of choices with the smallest cost. We are defining the selection of hardware checkers as a cost-based optimization problem. Specifically, we need to solve:

$$\min_{S}(\mathcal{F}(S))$$
$$\mathcal{PW}(S) \leq powerBudget, \quad \mathcal{AR}(S) \leq areaBudget$$

where S is a subset of checkers and $\mathcal{PW}$ and $\mathcal{AR}$ encode power and area constraints, respectively. $\mathcal{F}$ encodes the cost of the design and $\mathcal{F}(S)$ can depend on any of the power or area related costs together with any other considerations. In general, $\mathcal{F}(S)$ is non-linear (i.e., not a simple summation of cost for checkers) and depends on the subset of checkers that are implemented.

There are two major challenges in solving this optimization problem. The first problem is how to compute the functions $\mathcal{F}, \mathcal{PW}$ and $\mathcal{AR}$ given that there are 2^m possible inputs, where m is the number of checkers in the set S. This problem is difficult since estimating power or area for a given design requires expensive synthesis. Performing such an estimation for 2^m designs is infeasible. Our approach leverages machine learning techniques to treat the estimation problem as a regression problem. Instead of generating all possible designs, we will generate a small subset and learn estimates for area and power. Section 5.1 describes our cost prediction scheme.

The second problem is how to solve the optimization problem, i.e., how to find the set S that minimizes the cost while satisfying the constraints. This problem is difficult due to the size of the search space and the fact that the problem is non-linear. The non-linearity translates into a potentially large number of local minimums. To address the problem, we use non-linear optimization techniques. We need to adapt the techniques since we are optimizing over discrete sub-sets rather than metric spaces. Section 5.2 describes how our approach solves the optimization problem.

4 CLUSTERING-BASED ASSERTION SELECTION

Figure 2 shows an overview of our clustering-based assertion selection framework. It has three important steps: feature extraction, assertion clustering, and selection of dominating assertion from each cluster. The first step identifies beneficial features from assertions. The second step clusters similar assertions based on the similarity of the features. The final step selects one dominating assertion from each cluster to get the final minimized set of assertions. The remainder of this section describes these three steps in detail.

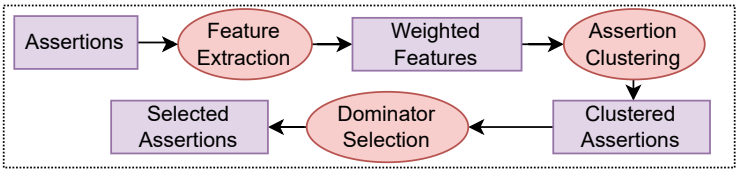


Fig. 2. Clustering-based assertion selection framework

4.1 Feature Extraction

To identify a dominating assertion, it is necessary to construct a set of clusters as outlined in Definition 6 using the initial assertions. These clusters are formed based on either full or partial redundancy in terms of covered non-vacuous failing states. In this section, we first present the mathematical proof of our proposed method for identifying assertions with full or partial redundancy in terms of covered non-vacuous failing states. Subsequently, we describe what should be used as features for the clustering algorithm and how to extract them to use the aforementioned mathematical method to perform clusters, which aims to minimize the number of assertions while ensuring significant fault coverage. To illustrate the key steps of this approach, we provide examples. For the following part of this section, let us assume that a_j, $a_k \in A$, and $P_1, P_2, P_3, and P_4$ are propositions.

Lemma 4.1. *If $P_1 \& P_2$ is true, then P_1 is true, where $P_1, P_2 \in U$.*

Theorem 4.2. $U_j^{at} \subseteq U_k^{at} \rightarrow B_k^{at} \subseteq B_j^{at}$

Proof. Assume that U_j^{at} is a subset of U_k^{at}, then $n(U_j^{at}) \leq n(U_k^{at})$, and $\forall P_{j,r}^{at} \in U_j^{at}$, $\exists!\ P_{k,t}^{at} \in U_k^{at}$ such that $P_{j,r}^{at} = P_{k,t}^{at}$. Therefore, we can express the antecedent of a_k in the form of $X \& Y$, where X is the antecedent of a_j and Y represents the combination of all propositions in $(U_k^{at} - U_j^{at})$ using the AND operator. Using the Lemma 4.1 ($X \& Y \rightarrow X$), we can conclude that whenever the antecedent of a_k becomes true, the antecedent of a_j becomes true. Therefore, B_k^{at} is a subset of B_j^{at}. □

Lemma 4.3. *If P_1 is false, then $P_1 \& P_2$ is false, where $P_1, P_2 \in U$.*

Theorem 4.4. $U_k^{ct} \subseteq U_j^{ct} \rightarrow B_k^{ct} \subseteq B_j^{ct}$

Proof. Assume that U_k^{ct} is a subset of U_j^{ct}, then $n(U_k^{ct}) \leq n(U_j^{ct})$, and $\forall P_{k,g}^{ct} \in U_k^{ct}$, $\exists!\ P_{j,h}^{ct} \in U_j^{ct}$ such that $P_{k,g}^{ct} = P_{j,h}^{ct}$. Therefore, we can express the consequent of a_j in the form of $Q \& R$, where Q is the consequent of a_k and R represents the combination of all propositions in $(U_j^{ct} - U_k^{ct})$ using the AND operator. Using the Lemma 4.3 ($\neg Q \rightarrow \neg(Q \& R)$), we can conclude that whenever the consequent of a_k becomes false, the consequent of a_j becomes false. Therefore, B_k^{ct} is a subset of B_j^{ct}. □

Theorem 4.5. $U_j^{at} \subseteq U_k^{at}$ *and* $U_k^{ct} \subseteq U_j^{ct} \rightarrow (B_k^{at} \cap B_k^{ct}) \subseteq (B_j^{at} \cap B_j^{ct})$

Proof. If $x \in (B_k^{at} \cap B_k^{ct})$, then $x \in B_k^{at}$ and $x \in B_k^{ct}$. According to Theorem 4.2, $B_k^{at} \subseteq B_j^{at}$, and therefore $x \in B_j^{at}$. According to Theorem 4.4, $B_k^{ct} \subseteq B_j^{ct}$, and therefore $x \in B_j^{ct}$. In other words, $x \in (B_j^{at} \cap B_j^{ct})$. Therefore, $\forall x \in (B_k^{at} \cap B_k^{ct})$, $x \in (B_j^{at} \cap B_j^{ct})$. Therefore, $(B_k^{at} \cap B_k^{ct})$ is a subset of $(B_j^{at} \cap B_j^{ct})$. □

Lemma 4.6. *If $(C \subseteq D)$ and $(C \subseteq E)$, then $(C \cap D \cap E \neq \emptyset)$, where C, D, and E are nonempty sets.*

According to Definition 11, Theorem 4.5, and Definition 7, the assertion with the fewest number of antecedent propositions and the most number of consequent propositions is the dominating assertion (a_d) among two assertions that have a relationship like in Theorem 4.5. This is because one assertion has full redundancy compared to covered non-vacuous failing states with the other. When finding a dominating assertion from a cluster of assertions, if we compare any two assertions in the cluster, they both should have properties as in Theorem 4.5. Then that cluster has a similarity property among elements as defined in Definition 6. Hence, according to Lemma 4.6, there should be at least one common proposition in the antecedent and at least one common proposition in the consequent in the cluster. Therefore, the assertion with the fewest number of antecedent

propositions and the most number of consequent propositions is the dominating assertion. This holds true when the assertions in the cluster have at least one common proposition in the antecedent and at least one common proposition in the consequent.

Table 1. Truth Table with four propositions

State	P_1	P_2	P_3	P_4
s0	0	0	0	0
s1	0	0	0	1
s2	0	0	1	0
s3	0	0	1	1
s4	0	1	0	0
s5	0	1	0	1
s6	0	1	1	0
s7	0	1	1	1

State	P_1	P_2	P_3	P_4
s8	1	0	0	0
s9	1	0	0	1
s10	1	0	1	0
s11	1	0	1	1
s12	1	1	0	0
s13	1	1	0	1
s14	1	1	1	0
s15	1	1	1	1

Table 2. Sample assertions and non-vacuous failing states

a_i	**Assertion**	**Non-vacuous failing states**	**# Non-vacuous failing states**
a_1	$P_1 \rightarrow P_2\&P_3\&P_4$	s8, s9, s10, s11, s12, s13, s14	7
a_2	$P_1 \rightarrow P_2\&P_3$	s8, s9, s10, s11, s12, s13	6
a_3	$P_1 \rightarrow P_2\&P_4$	s8, s9, s10, s11, s12, s14	6
a_4	$P_1 \rightarrow P_2$	s8, s9, s10, s11	4
a_5	$P_1\&P_3 \rightarrow P_2\&P_4$	s10, s11, s14	3
a_6	$P_1\&P_3 \rightarrow P_2$	s10, s11	2
a_7	$P_1\&P_4 \rightarrow P_2$	s9, s11	2
a_8	$P_1\&P_3\&P_4 \rightarrow P_2$	s11	1

For example, let us consider a set of assertions that contains four unique propositions: P_1, P_2, P_3, and P_4. Table 1 shows the truth table for all possible assertions using these four propositions. Note that this simple example is used to illustrate the theories. Nevertheless, these theories are valid for any common scenarios. Table 2 shows eight possible linear temporal logic assertions. By using Table 1, it is possible to find all the non-vacuous failing states for every assertion listed in Table 2. The third column of Table 2 represents respective non-vacuous failing states. It highlights the fact that the assertion with the fewest antecedent propositions and the most consequent propositions has more non-vacuous failing states, and it covers the non-vacuous failing states of all other listed assertions. According to Table 2, if we can find an assertion like a_1, it can cover all non-vacuous failing states that are covered by other assertions in the same cluster (Theorem 4.5). Thus, it has full redundancy with the failing states of all other assertions.

However, if an assertion like a_1 with full redundancy does not exist in the assertion cluster, then the next best options are a_2 or a_3 with partial redundancy. However, choosing assertions other than a_1 may result in uncovered non-vacuous failing states based on the set of assertions. For example, if a_2 is selected and a_3 or a_5 exists in the assertions cluster, the uncovered non-vacuous failing state is state s14. Suppose we expand the a_2, a_3 and a_5 assertions (using the Boolean tautology: $p = (p\&q)|(p\&\neg q)$, where p and q are Boolean propositions) as follows:

$$a_2 : (P_1\&P_3)|(P_1\&\neg P_3) \rightarrow (P_2\&P_3\&P_4)|(P_2\&P_3\&\neg P_4)$$
$$a_3 : (P_1\&P_3)|(P_1\&\neg P_3) \rightarrow (P_2\&P_3\&P_4)|(P_2\&P_4\&\neg P_3)$$
$$a_5 : (P_1\&P_3) \rightarrow (P_2\&P_3\&P_4)|(P_2\&P_4\&\neg P_3)$$

Some common propositions can be found in the antecedent and the consequent parts of expanded a_2, a_3, and a_5 assertions. If the antecedent of a_5 becomes true, the antecedent of a_3 also becomes true because they have a common antecedent proposition, in this case, $(P_1 \& P_3)$. If the consequent of a_5 becomes false, the consequent of a_3 also becomes false because they have common consequent propositions, in this case, $(P_2 \& P_3 \& P_4) \mid (P_2 \& P_4 \& \neg P_3)$. Thus, a_5 can be removed from the cluster while keeping a_3.

The same analysis can be applied to assertions a_2 and a_3. Between a_2 and a_3, if the antecedent of one of them becomes true, the antecedent of the other also becomes true. However, in the consequent part, even though both assertions have a common proposition, it is not guaranteed that every time the consequent of a_2 becomes false, the consequent of a_3 also becomes false. The reason for this is that both assertions have unique propositions in the consequent that are not common for both, in this case, $(P_2 \& P_3 \& \neg P_4)$ in a_2 and $(P_2 \& P_4 \& \neg P_3)$ in a_3. Thus, some non-vacuous failing states are not common for both assertions. If we choose either a_2 or a_3, there are some uncovered non-vacuous failing states. During our experimental evaluation (Section 6), we show that the reduction in the covered non-vacuous failing states only has a minor impact. Therefore, an efficient way for the minimization is to create clusters where each assertion in the cluster comprises at least one common antecedent proposition and at least one common consequent proposition.

To implement the above clustering method, the feasible features are the unique propositions in the antecedent and consequent, which are the basis propositions of the given set of assertions. While this feature extraction is promising, it is not practical to always find clusters that satisfy both of the following conditions: (1) every assertion in the cluster must have at least one common proposition in the antecedent, and (2) every assertion in the cluster must have at least one common proposition in the consequent. It is possible to modify the above two conditions to give more flexibility to the clustering algorithm while maintaining the fault coverage. This enables us to explore either common antecedent-based clustering or common consequent-based clustering.

We evaluate the applicability of antecedent-based versus consequent-based clustering using five benchmarks. Table 3 shows that for every benchmark, there are only a few unique propositions in the consequent compared to the antecedent. Therefore, there are more assertions with the same consequent proposition than assertions with the same antecedent proposition. Thus, clusters based on common consequent propositions contain more similar assertions in every cluster than clusters based on common antecedent propositions. This allows for minimizing the initial assertion set by a considerable amount while maintaining significant fault coverage. Therefore, we select common consequent-based clustering, where every assertion in the cluster has at least one common proposition in the consequent.

To implement consequent-based clustering, we used a higher weight ($e = 2$) for propositions in consequent than in the antecedent ($e = 1$) as feature values. This weighing method reflects the importance of the propositions in the consequent when considering them as feature values for clustering. Lines 2 - 18 of Algorithm 1 show the major steps of the above-mentioned feature extraction, which produces extracted quantified feature array (E) as the output. The algorithm begins by constructing two sets: AN and CO. The set AN contains all the unique propositions found in the antecedents of each assertion in the set A. Similarly, the set CO contains all the unique propositions found in the consequent of each assertion in the set A. These two sets are used to quantify the features of each assertion to produce the set of extracted features (E).

4.2 Assertion Clustering

Clustering is an unsupervised learning technique that groups similar data points based on features. There are several clustering algorithms, such as K-means clustering, hierarchical clustering, density-based clustering, etc. BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies) [26] is

Table 3. Unique propositions in antecedent and consequent using five benchmarks: *Arbiter* (arb), *Ibex Compressed Decoder* (ibex_dec), *Ibex Controller* (ibex_con), *Ibex Id-Stage* (ibex_id), *Ibex Multdiv Slow* (ibex_mul)

Benchmark	# Assertions	# Unique Propositions in Antecedent	# Unique Propositions in Consequent	# Overlapping Propositions
arb	23	12	8	0
ibex_dec	1499	191	79	0
ibex_con	6602	1090	97	0
ibex_id	6447	1972	817	4
ibex_mul	2648	855	200	0

Algorithm 1: Clustering-Based Assertion Selection

Input : Set of assertions A
Output: Dominating assertions set S

```
/* Feature Extraction */
AN ← UniqueAntecedentPropositions(A);
CO ← UniqueConsequentPropositions(A);
foreach a_u ∈ A do
    foreach proposition P_w in a_u do
        if (P_w ∈ CO) then
            e ← 2
        end
        else if (P_w ∈ AN) then
            e ← 1
        end
        else
            e ← 0
        end
        E_v ← E_v || e
    end
    E ← E ∪ E_v
end
/* Assertions Clustering */
CL ← Clustering(A, E);
/* Dominator Selection */
foreach A_v ∈ CL do
    D' ← MinimumAntecedentAssertions(A_v, E);
    D ← MaximumConsequentAssertion(D', E);
    S ← S ∪ D;
end
```

a hierarchical clustering algorithm designed for large-scale datasets. We used BIRCH clustering because it is more flexible in capturing clusters with varying shapes and densities and its hierarchical structure allows for a better representation of complex clusters, including those with varying densities or irregular shapes. In our assertion selection method, input features for the clustering algorithm fully depend on the set of input assertions (A). BIRCH comprises a noise-handling mechanism called the "noise points" in its Clustering Feature tree (CF-tree). These points help capture and isolate noisy or outlier data, preventing them from significantly affecting the clustering

process. Line 20 of Algorithm 1 depicts the clustering function of our proposed framework, which takes A, and E as inputs. The output of this function is the set of assertion clusters (CL). As explained in Section 4.1, the weight-encoded unique proposition feature array (E) assists the clustering algorithm in identifying assertions with similar propositions, especially in the consequent. It then produces the set of assertion clusters (CL) in a way that each cluster in CL contains assertions with at least one common consequent proposition. One major advantage of this minimization method, compared to existing methods such as [19], is that by adjusting the number of clusters, we can control the final number of assertions in the minimized assertion set.

4.3 Dominator Selection

Line 22 - 26 of Algorithm 1 is used to select the dominating assertion from the assertion clusters (CL). Line 23 ($MinimumAntecedentAssertions$) returns the set of assertions with the least number of propositions in the antecedent by using the formed assertion clusters. As outlined in Section 4, a smaller number of propositions in the antecedent is better because it has relatively high non-vacuous failing states coverage. Line 24 ($MaximumConsequentAssertion$) finds the most suitable assertion with the most number of propositions in the consequent from the remaining assertions in the cluster. If there is more than one suitable assertion, the function randomly selects an assertion from suitable assertions. The final set of selected assertions consists of one profitable assertion from each cluster.

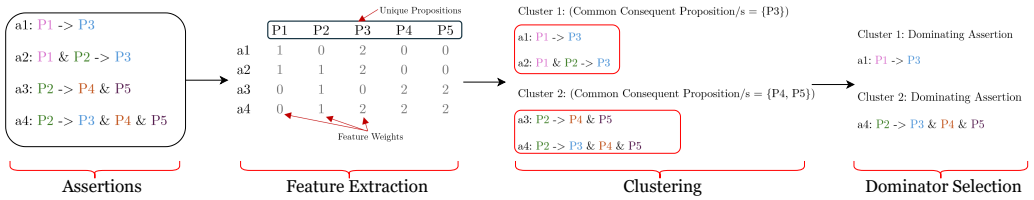


Fig. 3. Example for clustering-based assertion minimization

Figure 3 illustrates a full example of the proposed assertion minimization method. Initially, there are four assertions in the set. After identifying the unique propositions ($P1, P2, P3, P4, P5$), we generate the input feature array for the clustering process. In this array, '1' is assigned when a unique proposition is in the antecedent part ($P1$ is in the antecedent of $a1$), '2' is assigned when a unique proposition is in the consequent part ($P3$ is in the consequent of $a1$), and '0' is assigned when a unique proposition is not in the assertion ($P2$ is not in $a1$). Utilizing these weighted features, we perform common consequent-based clustering (Cluster 1 and Cluster 2). Finally, we select the dominating assertion from each cluster ($a1$ from Cluster 1). This assertion has the least number of propositions in the antecedent and the most number of propositions in the consequent.

In the process of finding dominating assertions from a set of multi-cycle assertions, which contain time delays, Boolean expressions in both the antecedent and consequent are converted to a new type of Boolean expression that directly indicates its time with respect to the current clock cycle, as described in Definition 4. Consider two assertions: $a_9 : P1 \,\#\#2\, P2 \rightarrow P3 \,\&\, P4$ and $a_{10} : P1 \,\#\#1\, P1 \,\#\#1\, P2 \rightarrow P3$. If the assertion a_9 is triggered, it means proposition $P2$ is true and propositions $P3$ or $P4$ are false in the current clock cycle, while proposition $P1$ was true two clock cycles before. Then assertion a_9 is converted to a different naming convention by removing the delay expression '##2' while maintaining the assertion functionality. Therefore, $a_9 : P1 \,\#\#2\, P2 \rightarrow P3 \,\&\, P4$ is converted to $a_9 : P1[-2] \,\&\, P2 \rightarrow P3 \,\&\, P4$, where $P1[-2]$ indicates that this assertion checks the value of $P1$ two clock cycles before. Furthermore, $P2$, $P3$, and $P4$ remain unchanged because the

assertion checks their values in the current clock cycle. Note that when unique Boolean expressions are being extracted, $P1$ and $P1[-2]$ are considered two different Boolean expressions.

Table 4. Sample assertions and propositions states with time

	Clock Cycle					
Propositions	1	2	3	4	5	6
$P1$	1	1	1	0	0	1
$P1[-1]$	x	1	1	1	0	0
$P1[-2]$	x	x	1	1	1	0
$P2$	0	1	1	1	1	0
$P3$	0	1	0	1	0	0
$P4$	0	0	0	0	1	0
Assertions	1	2	3	4	5	6
a_9	0	0	1	1	1	0
a_{10}	0	0	1	0	0	0

For assertions a_9 and a_{10}, five unique propositions can be identified: $P1[-2]$, $P1[-1]$, and $P2$ from the antecedent parts, and $P3$ and $P4$ from the consequent parts. $P1[-2]$ and $P2$ are common antecedent parts, and $P3$ and $P4$ are common consequent parts. Table 4 shows an example snapshot of proposition states and assertion states over time. Values for $P1$, $P2$, $P3$, and $P4$ are chosen randomly. In Table 4, 1 and 0 represent the 'true' and 'false' states for propositions and the 'failing' and 'passing' states for assertions, respectively. Here, 'failing' means 'non-vacuous failing state'. The symbol 'x' represents an unknown state. The table clearly shows that the proposed method still identifies the dominating assertion, in this case, assertion a_9, which has the most number of consequent propositions and the least number of antecedent propositions.

5 DYNAMIC REFINEMENT OF HARDWARE CHECKERS

The assertions selected in the previous section are considered as inputs for dynamic refinement. Figure 4 shows an overview of our dynamic refinement framework. It has two important steps: (1) cost prediction and (2) optimization. The first step is to learn how to predict cost for a given set of hardware checkers. The second step uses the trained model to find the optimal set of hardware checkers that satisfies the constraints while minimizing the cost of adding the hardware checkers. The remainder of this section describes these two steps in detail.

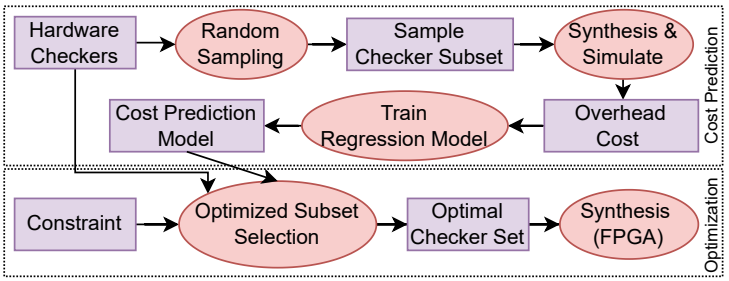


Fig. 4. Overview of our dynamic refinement scheme

5.1 Cost Prediction

In this section, we address the problem of estimating the functions $\mathcal{F}, \mathcal{PW}$ and $\mathcal{AR}$ that appear in the optimization problem in Section 3.3. Typically, the function $\mathcal{F}$ is a simple cost model depending on the power and area functions $\mathcal{PW}(S)$ and $\mathcal{AR}(S)$, respectively, when implementing assertions in the set S. While the two problems (power and area) capture different aspects of circuit design, the estimation problem is essentially the same. We solve the estimation problem by modeling the problem as a regression problem with sets as inputs: Given samples $S_1, \ldots, S_g$ and power consumption estimates $p_1 = \mathcal{PW}(S_1), \ldots, p_g = \mathcal{PW}(S_g)$ find a good approximation of the function $\mathcal{PW}(S)$. The same solution can be used for $\mathcal{AR}$ as well.

Most of the regression models in machine learning and statistics literature only accommodate continuous inputs. Essentially, the regression models find non-linear mapping from $\mathbb{R}^g \longrightarrow \mathbb{R}$. To finish our translation of the cost estimation problem into a regression problem, we transform the set input S into a continuous input by introducing an input i for the regression problem for each checker C_i. Next, we set the input value at 0.0 if $C_i \notin S$ and at 1.0 if $C_i \in S$. Thus, each set S is always mapped into a vector of size m that contains only 0.0 or 1.0 entries.

As shown in line 2 of Algorithm 2, random sampling is conducted on hardware checkers (S) to get different sample subsets (S'). Line 3 uses these sample subsets, synthesizes and simulates them to determine the overhead cost (O) in terms of power and area. The overhead costs are used to train the regression models in line 4 and in line 5. Once learning models ($\mathcal{PW}(S)$ and $\mathcal{AR}(S)$) are built, they can be used to predict $\mathcal{F}(S)$ simply by encoding the input S using the same 0.0, 1.0 mapping and using the predictor for the estimate.

5.2 Optimization

When non-linear optimization problem is relatively simple (i.e., it has a small number of local minimums), gradient descent methods perform fairly well. Our optimization problem is likely to be complicated due to the interaction between the hardware checkers. In such situations, simulated annealing methods are preferred. The proposed algorithm aims to combine the advantages of both gradient descent and simulated annealing. Since the proposed algorithm needs to be used in hardware to find suitable hardware checkers in real-time, it must be fast, therefore, we chose gradient descent. In order to escape local minima, we utilize simulated annealing. The random steps are allowed more often in the beginning but less and less often as the computation progresses so the solution finds a better local minimum. In our algorithm, a coin flip is used to switch between gradient descent and simulated annealing. Instead of choosing the worst solution in simulated annealing, we use a cooling mechanism to switch between gradient descent and simulated annealing. As a result, we utilize gradient descent with simulated annealing to solve the non-linear optimization problem.

Line 9 - 27 of the Algorithm 2 presents a gradient descent search of the subset space with simulated annealing. Inputs of the algorithm are $\mathcal{F}(S)$ and $C(S)$. Function $\mathcal{F}(S)$ is retrieved using the cost prediction model described in Section 5.1. Function $C(S)$ represents a Boolean function that combines all the constraints and indicates whether the constraints are satisfied for S or not. The results of this function will be a locally optimal solution S, which satisfies $C(S)$. The algorithm begins at a random initial feasible subset (lines 9 - 11). First, the probability of the random step is given value 0.5 (line 12). Then, this probability is halved through the annealing loop (lines 13 - 27). The loop is conducted for 'Max' steps by checking whether the probability is greater than 'Prob'. In each iteration, 'FlipCoin' is conducted to determine whether we will move to the best feasible neighboring subset (lines 22 - 24) or will move to a random feasible neighboring subset (lines 19 - 21). A neighboring subset of S is one which has only one hardware checker added or removed relative to S. A subset is feasible when it satisfies the applied constraints. Function 'FlipCoin' will

Algorithm 2: Dynamic Refinement of Checkers

Input : Initial hardware checkers S, Max, Prob
Output : Locally optimal solution S

1 /* COST PREDICTION */;
2 $S' \leftarrow RandomSamples(S)$;
3 $O \leftarrow OverheadCalculator(S')$;
4 $\mathcal{PW}(S) \leftarrow PowerRegressionModel(O, S')$;
5 $\mathcal{AR}(S) \leftarrow AreaRegressionModel(O, S')$;
6 $S'' \leftarrow Encoder(S)$;
7 $\mathcal{F}(S) \leftarrow CostPredictor(\mathcal{PW}(S), \mathcal{AR}(S), S'')$;
8 /* OPTIMIZATION */;
9 **repeat**
10 | select random S
11 **until** $C(S)$ *is true*;
12 $p \leftarrow 1/2$;
13 **while** $p > Prob$ **do**
14 | Done $\leftarrow$ False;
15 | Steps $\leftarrow$ 0;
16 | **while** $\neg$ *Done* & *Steps* < *Max* **do**
17 | | Steps $\leftarrow$ Steps + 1;
18 | | Coin $\leftarrow$ FlipCoin(p);
19 | | **if** *Coin == Head* **then**
20 | | | i $\leftarrow$ random(1,n);
21 | | | $S \leftarrow S \oplus C_i$;
22 | | **else**
23 | | | Done $\leftarrow GradientDescent(\mathcal{F}(S), C(S), S)$
24 | | **end**
25 | **end**
26 | $p \leftarrow p/2$;
27 **end**

allow more random steps in the beginning (i.e., when p is large) but less random steps when p is smaller. For the best feasible solution, the gradient descent function *GradientDescent* is used (line 23). The probability of a random move decreases by a factor of 0.5 every 'Max' steps (line 26). The algorithm stops when it attempts to move to the best neighboring subset, and no feasible neighbor is superior to the current solution.

There are several aspects that can change the parameters of the cost-based optimization problem. One is the value of including an assertion can shift significantly throughout the life-cycle. Assertions that seem marginal now can become very important due to discoveries of new functional exploits. Similarly, assertions that seem important now, can prove to have only marginal benefits in the future. Either the number of samples, the quality of the synthesis estimation or the learning methods can improve throughout the life-cycle. Another aspect is that if more computation can be afforded, it can result in better quality solutions for the optimization problem.

Based on our formulation and solution for the cost-based optimization problem, a number of shortcuts can be taken to improve the running time of the solver. Specifically, the current best solution can be used as the starting point for the modified future optimization problem. It is likely

that the problem will not shift significantly. The current solution should be in the neighborhood of the new optimal solution.

Table 5. Benchmarks details

Benchmark	HARM		Goldmine		Total Assertions	#Faults	#Lines	I/O	#Checkers	#Samples
	#Assertions	Coverage	#Assertions	Coverage						
arb	17	100.00%	6	58.33%	23	12	28	6	6	50
ibex_dec	933	100.00%	566	100.00%	1499	848	97	4	20	1000
ibex_con	5480	30.99%	1122	24.41%	6602	926	459	57	10	300
ibex_id	5374	83.23%	1073	36.43%	6447	1002	484	82	20	1000
ibex_mul	2212	69.19%	436	3.13%	2648	1980	230	15	10	300

6 EXPERIMENTS

In this section, we evaluate the effectiveness of our proposed approaches. First, we describe our experimental setup. Next, we outline the results of our experiments.

6.1 Experimental Setup

For the experimental evaluation, we have selected benchmarks from [27] as shown in Table 5. The first column of the table shows the benchmarks. The second and third columns show the number of assertions mined using HARM [9] and their fault coverage, respectively. Similarly, the fourth and fifth columns represent the number of assertions mined using Goldmine [10] and their fault coverage, respectively. The sixth column presents the total number of assertions used for the clustering method. The seventh column shows the total number of inserted faults. The eighth and ninth columns show the number of code lines in the design and the total number of primary inputs/outputs respectively. The tenth column presents the number of assertions selected as hardware checkers for synthesis. The last column shows the number of samples used for the cost prediction model.

Table 6. Example assertions generated for the *Ibex Controller* benchmark

Assertions
$!csr\_mstatus\_tw\_i$ ##1 $stall\_jump\_i$ & $illegal\_insn\_i$ & $priv\_mode\_i[0]$ ##2 $!pc\_id\_i[26]$ & $!instr\_compressed\_i[6] \rightarrow !csr\_mtval\_o[12]$
$!store_err\_d$ & $!special\_req$ ##1 $!ebrk\_insn\_i$ & $!fetch\_enable\_i$ ##2 $instr\_i[2] \rightarrow !csr\_mtval\_o[29]$
$!instr\_fetch\_err$ & $!special\_req$ ##3 $!instr\_compressed\_i[8]$ & $!lsu\_addr\_last\_i[29]$ & $!pc\_id\_i[22] \rightarrow !csr\_mtval\_o[18]$
$csr\_pipe\_flush\_i$ ##1 exc_req_q ##2 $!instr\_compressed\_i[8]$ & $!lsu\_addr\_last\_i[29]$ & $!instr\_i[19]$ & $lsu\_addr\_last\_i[16] \rightarrow !csr\_mtval\_o[18]$
$ecall\_insn$ ##1 $!dret\_insn$ & $!store\_err\_q$ ##2 $!instr\_compressed\_i[8]$ & $!lsu\_addr\_last\_i[29]$ & $!lsu\_addr\_last\_i[16] \rightarrow !csr\_mtval\_o[18])$

In the experiment setup, assertions are generated using HARM [9] and Goldmine [10], and total assertions of HARM and Goldmine are used for the clustering method. All assertions generated using HARM and Goldmine belong to linear temporal logic. To ensure a fair evaluation of the methods, we conducted experiments with the maximum total delay in the assertion set to three cycles, the maximum number of propositions in the antecedent set to six, and all variables in the assertions restricted to Boolean type. Table 6 presents some example assertions generated for the Ibex Controller benchmark. Here, we use fault coverage as the evaluation metric. To avoid any bias, assertion generation was performed before the insertion of vulnerabilities. Although HARM has the ability to find the minimum set of assertions that covers all the given faults, for these experiments, we assume the faults are unknown, and we evaluate the generated and minimized set of assertions based on their covered non-vacuous failing states. Therefore, a set of faults are inserted, which consist of (1) “bit flip", where a bit of a selected variable is flipped and (2) “functional fault", where the Boolean operation of a selected Boolean expression is changed with another Boolean operation. We selected 75% of the variables randomly and performed bit flip. In order to introduce functional faults, we randomly selected 20% lines (statements) in the Verilog designs and changed an operation

in that line. All the fault-inserted benchmarks are simulated for 1000 clock cycles to generate the trace files. These trace files are used to identify faults that are covered by each assertion, and that information is used to calculate fault coverage for different types of assertion sets. We count the number of faults that are covered by the selected set of assertions. Fault coverage is calculated by expressing the number of covered faults as a percentage of the total number of inserted faults. Note that assertion generation as well as assertion selection was performed prior to fault injection. In other words, our clustering method did not have any knowledge of inserted faults.

We have carried out the clustering-based assertion selection using a 2.9 GHz 8-core AMD Ryzen 7 4800HS processor equipped with 16 GB of RAM. We have evaluated the dynamic refinement of hardware checkers using the Zynq-7000 SoC based evaluation platform. We synthesized the original design as well as the design with embedded assertions, to the FPGA in the SoC. We utilized Xilinx Vivado Design Suite 2021 to perform synthesis, optimization, and place and route for the designs. The same software was then used to perform timing-accurate simulation of the FPGA mapped designs, emulating both the functional and timing constraints of the FPGA architecture.

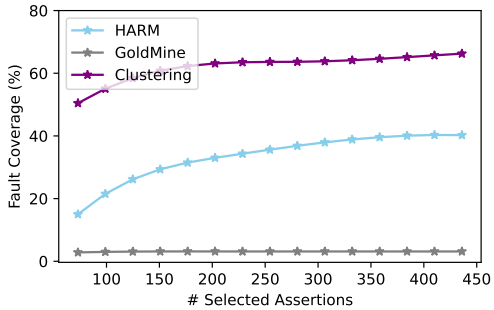


Fig. 5. Fault coverage results for ibex_mul from Table 7

6.2 Assertion Selection Results

Table 7 reports the fault coverage results for the five benchmarks outlined in Table 5. The first column shows the benchmark name. The second column represents the number of selected assertions, which is the number of selected ranked assertions from the top in HARM and Goldmine, and the number of clusters in the clustering method. For example, if the number of selected assertions is 73, all of the following conditions are true: (i) we have selected the top 73 assertions from the HARM ranked assertions, (ii) we have selected the top 73 assertions from the Goldmine ranked assertions, and (iii) we have considered 73 clusters in the clustering method to select 73 assertions. The last three columns show the fault coverage of HARM, Goldmine, and the clustering method, respectively. We indicate the coverage as a percentage of the number of inserted faults.

The number of mined assertions using Goldmine is less than HARM. Therefore, we use the number of Goldmine assertions as the upper bound for the number of selected assertions. Then we divide that number into six subsets to analyze the fault coverage. It clearly shows that the clustering-based assertion selection consistently provides higher fault coverage than HARM and Goldmine. For example, in the ibex_mul benchmark, the maximum number of selected assertions is 436 (number of assertions generated using Goldmine). We set that number as the upper bound and divide the number of assertions space into six almost equal sets. Therefore, each row increment represents a 16.67% increment in the number of assertions. The fault coverage is calculated for selected assertions, and it has been represented as a percentage of the total number of inserted faults. Our clustering method consistently outperforms both HARM (78% on average) and Goldmine (6 times on average) in terms of fault coverage. Figure 5 provides a visualization of the fault coverage

Table 7. Comparison of assertion selection results. Here we compare the fault coverage among ranked assertions obtained from the top in HARM and Goldmine, alongside assertions acquired through our clustering method.

Benchmark	# Selected Assertions	Fault Coverage		
		HARM	Godmine	Clustering
arb2	1	33.33%	8.33%	41.67%
	2	33.33%	33.33%	50.00%
	3	33.33%	58.33%	100.00%
	4	33.33%	58.33%	100.00%
	5	100.00%	58.33%	100.00%
	6	100.00%	58.33%	100.00%
ibex_dec	94	100.00%	5.66%	100.00%
	189	100.00%	8.84%	100.00%
	283	100.00%	14.15%	100.00%
	377	100.00%	15.68%	100.00%
	472	100.00%	16.98%	100.00%
	566	100.00%	100.00%	100.00%
ibex_con	187	14.79%	9.61%	30.78%
	374	16.74%	18.79%	35.31%
	561	17.17%	22.57%	36.39%
	748	17.49%	23.87%	36.50%
	935	19.01%	23.87%	36.83%
	1122	19.01%	24.41%	36.83%
ibex_id	179	14.77%	8.58%	31.14%
	358	25.15%	18.06%	57.98%
	536	32.53%	22.95%	70.76%
	715	39.12%	28.64%	75.85%
	894	41.52%	32.73%	79.94%
	1073	52.79%	36.43%	80.24%
ibex_mul	73	15.00%	2.83%	50.45%
	145	28.74%	3.13%	60.35%
	217	33.74%	3.13%	63.38%
	290	37.27%	3.13%	63.69%
	363	39.70%	3.13%	64.70%
	436	40.25%	3.13%	66.26%

results of Table 7 for the ibex_mul benchmark. As shown in Figure 5, our clustering method is able to provide higher coverage with fewer assertions compared to Goldmine and HARM. The figure also highlights that increasing the number of assertions does not significantly enhance fault coverage.

Table 8 shows fault coverage results for assertion minimization of the top-ranked HARM assertions. Our objective is to determine if we can reduce the number of top ranked HARM assertions with negligible impact on fault coverage. We use the same five benchmarks as shown in the first column. The second column shows the number of selected HARM assertions, and the third column shows the corresponding fault coverage. The fourth and fifth columns show the number of assertions after the minimization using our clustering method and corresponding fault coverage, respectively. The sixth column shows the reduction in the number of assertions as a percentage of the initial number of HARM assertions. The final column shows the loss in the fault coverage due to reduction in assertions. We can observe that our clustering method can significantly (50%

on average) reduce the top-ranked HARM assertions with minor loss (2.51% on average) in fault coverage.

Table 8. Fault coverage comparison using assertions generated by HARM. Here we reduce the number of top-ranked HARM assertions by 50% using the proposed clustering method.

Benchmark	HARM		Clustering		% Reduction in Assertions	Loss in Fault Coverage
	# Assertions	Fault Coverage	# Assertions	Fault Coverage		
arb	2	33.33%	1	33.33%	50.00%	0.00%
ibex_dec	189	100.00%	94	100.00%	50.26%	0.00%
ibex_con	374	16.74%	187	14.90%	50.00%	10.97%
ibex_id	358	25.15%	179	25.15%	50.00%	0.00%
ibex_mul	145	28.74%	73	28.28%	49.66%	1.58%

Table 9. Fault coverage comparison for the final set of hardware checkers considering design constraints.

Benchmark	# Selected Assertions	Fault Coverage		
		HARM	Godmine	Clustering
arb2	6	100.00%	58.33%	100.00%
ibex_dec	20	100.00%	2.24%	100.00%
ibex_con	10	0.86%	3.46%	6.37%
ibex_id	20	0.00%	2.10%	6.29%
ibex_mul	10	3.38%	2.78%	16.57%

When it comes to post-silicon validation and debug, the number of hardware checkers is determined by design constraints such as area and power. For example, we selected only 6 assertions for the smallest (arb) benchmark, while we allowed 20 assertions for a larger (ibex_dec) benchmark. Table 9 shows the reduced number of assertions for different benchmarks. As expected, our proposed clustering method outperforms HARM and Goldmine generated assertions. We used these reduced set of assertions as the hardware checkers to evaluate cost prediction and dynamic refinement in the subsequent sections.

6.3 Cost Prediction Results

To emphasize that our cost function $\mathcal{F}(S)$ is not simply linear, we conducted an experiment where we calculated the power consumption of individual checkers and the power consumption of number of the checkers together. Figure 6 presents the power consumption (in watts) for different number of checkers for ibex_con design. The figure shows the cumulative cost (addition of individual checkers) and the actual power consumption for the same number of checkers. The cumulative cost of the 10 individual checkers is 0.030 watts. However, when we get the 10 checkers together, the power consumption is 0.019 watts. This shows that our cost function $\mathcal{F}(S)$ is non-linear. Therefore, it is important to use cost prediction techniques to predict the cost rather than synthesizing all possible combinations of checkers.

The size of Lookup Tables (LUT) and power overhead are selected as parameters for the cost prediction framework. To enable approximate prediction of the cost function, we have explored the efficacy of four models: (1) linear regression (LR), (2) linear regression with quadratic interaction (LRQ), (3) linear regression with cubic interaction terms (LRC), and ridge regression (RR). The training data was generated by collecting LUT and power data from a random sample of subsets from the set of all possible subsets for each design. Each assertion subset in the sample was then synthesized, optimized, placed, and routed. Hardware and power utilization data for each subset in the sample was then dumped, which would serve as the training data. For example, in case of

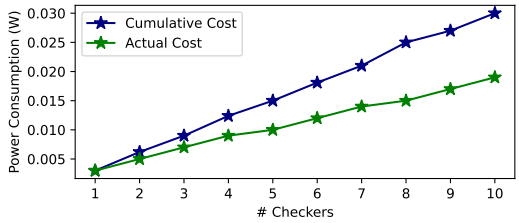
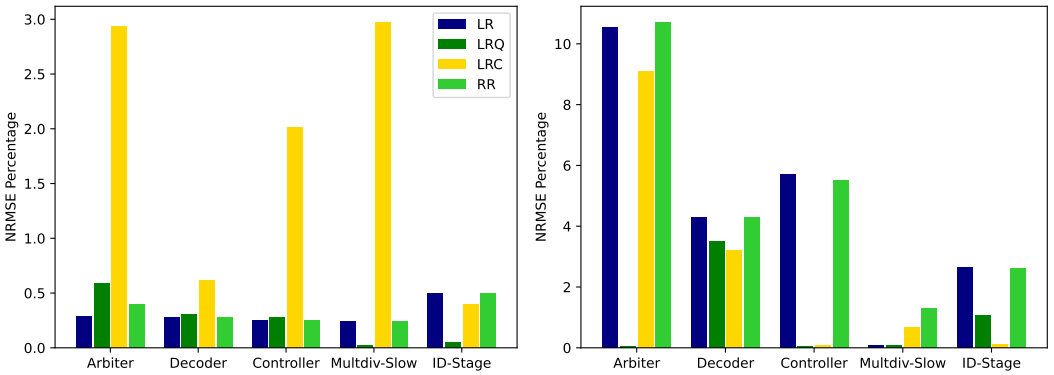


Fig. 6. Power consumption (W) for checkers in ibex_con

ibex_con, a sample of 300 subsets was collected, covering 29.29% of all possible subsets. Similarly, in case of ibex_dec, 1000 subsets were collected, covering 0.95% of all possible subsets. This data set was then randomly partitioned into a train and test set, using an 80-20 train-test split. First, the performance of the models on unseen data was estimated by training and evaluating the models on the train set using 10-fold cross validation, with normalized root mean squared error (NRMSE) as the performance metric. The performance evaluation for all four models for each benchmark with respect to power and LUT consumption is shown in Figure 7. For all the designs, the four models achieved less than 3% error predicting the power consumption and less than 5% error predicting LUT.

The model with the best performance on the test set for each metric and design pair (Figure 7) was then utilized during the subset selection algorithm in dynamic refinement to predict the LUT and power overhead for potential selected hardware checker subsets. By estimating whether a checker subset satisfied the applied constraints, the subset selection algorithm produced results without the processing bottleneck of HDL synthesis for each subset and its neighbors.



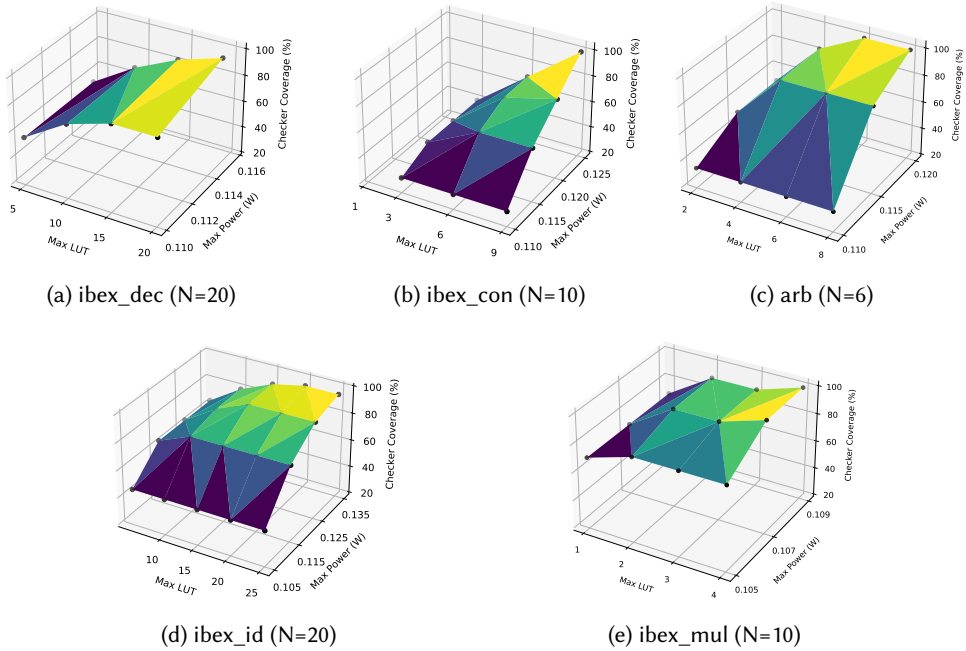
(a) Model accuracy for power consumption prediction (b) Model accuracy for LUT prediction

Fig. 7. Model accuracy for different regression models

6.4 Dynamic Refinement Results

The subset selection algorithm was run for all designs with a variety of parameters, including LUT constraints, power constraints, and number of iterations (by changing the 'prob' value). Figure 8 shows the dynamic refinement of hardware checkers for each benchmark with different constraints. For each benchmark, the number of checkers selected as optimal subset is shown in checker coverage as a percentage of all the number of checkers in the initial set (N). The dynamic refinement is performed with increasing iterations from 5 to 100 for all the constraints pairs. Figure 8 shows

the optimal selection of hardware checkers for each constraints pair chosen from the results from different iterations.



(a) ibex_dec (N=20) (b) ibex_con (N=10) (c) arb (N=6)

(d) ibex_id (N=20) (e) ibex_mul (N=10)

Fig. 8. Checker subset selection for changing requirements

When the design constraints are loosened, the achievable assertion coverage increases. For all the designs, the highest coverage values are achieved with maximum LUT of 45 and power of 0.155 watts. As the constraints are tightened, the coverage value tends to decrease as the algorithm must sacrifice coverage for feasibility. The loss in coverage for each decreasing step in one constraint value is not linear. Instead, the coverage begins to decrease more rapidly as the constraints decrease to values significantly below the mean value of the metrics for random subsets. This may be indicative of the fact that the propensity of the algorithm to find locally optimal, but globally sub-optimal solutions increases as the constraints are tightened due to the local search being constrained by a high amount of infeasible neighbors. It also may arise from the underlying distribution of subset overhead values being non-uniform. In other words, the fraction of subsets satisfying the tight constraints is lower than we would expect from a uniform distribution.

Table 10. Dynamic refinement results for ibex_con with different iterations

LUT	Power(W)	Iterations = 5	Iterations = 10	Iterations = 20	Iterations = 40	Iterations = 100
3	0.110	“0001000011" (3) P=0.10984, LUT=2.99	“0010100010" (3) P=0.10987, LUT=2.99	“0001000011" (3) P=0.10984, LUT=2.99	“0000100011" (3) P=0.10982, LUT=2.00	“0000100011" (3) P=0.10982, LUT=2.00
6	0.115	“0011011011" (6) P=0.11493, LUT=6.00	“0011011011" (6) P=0.11493, LUT=6.00	“0110101011" (6) P=0.11493, LUT=4.99	“0010111011" (6) P=0.11491, LUT=5.00	“0010111011" (6) P=0.11491, LUT=5.00
9	0.120	“0110111111" (8) P=0.11836, LUT=7.00	“0111111011" (8) P=0.11836, LUT=6.99	“0111111011" (8) P=0.11836, LUT=6.99	“0111111011" (8) P=0.11836, LUT=6.99	“0111111011" (8) P=0.11836, LUT=6.99

The results of running the dynamic refinement algorithm for ibex_con benchmark with increasing iterations (5 to 100) are shown in Table 10. The first two columns provide design constraints in terms of upper limit on the number of LUTs available and power consumption (in Watts). Each

row represents a different configuration in terms of constraints. The number of checkers selected for synthesis is shown in brackets. The solution subset is presented as a bit-string, where the i-th bit being 1 implies that the i-th checker was included in the solution. For each solution, the power and LUT values are also shown in the table. Note that significant increase in the number of iterations does not dramatically improve the achieved coverage. For example, the algorithm achieved a coverage of 8 (80.00%) with optimal solution even from 10 iterations for LUT=9 and Power = 0.120 W on the ibex_con design. However, for some constraints increasing the number of iterations helped to achieve the optimal solution (LUT=3 and Power=0.110).

Overall, the subset selection algorithm allowed for consistent performance under constraints in achievable ranges, and did not require significant iterations to find satisfactory solutions. The algorithm's speed and relative simplicity are positive indicators of the potential efficacy of this method in the dynamic refinement of on-chip security assertions. Our proposed algorithm represents a highly extensible foundation which can be augmented with additional constraints, such as novel cost functions, and time dependent behaviors, all of which potentially appear in industrial applications.

7 CONCLUSION

Post-silicon validation relies on observability infrastructure such as trace buffers. Hardware checkers can improve the observability for debugging functional as well as non-functional (e.g., security) violations. Due to hardware overhead considerations, it is not feasible to map all pre-silicon assertions as post-silicon hardware checkers. While there are promising approaches for selecting a small set of profitable assertions for synthesis, they are inefficient under a vast number of pre-silicon assertions and changing workloads and input variations. We developed efficient methods for feature selection, assertion clustering, and dominator selection to enable clustering-based assertion minimization while maintaining considerable fault coverage. We have utilized the selected pre-silicon assertions as inputs for post-silicon dynamic refinement. Specifically, we formulated the dynamic refinement problem as a cost-based non-linear optimization problem. We used regression-based learning to perform cost prediction for hardware checkers. We solved the non-linear optimization problem using gradient descent with simulated annealing. Our experimental evaluation demonstrated the effectiveness of assertion minimization as well as dynamic refinement. Specifically, our clustering method provided significantly higher fault coverage compared to top-ranked HARM (78% on average) and Goldmine (6 times on average) assertions. Similarly, our cost prediction is reasonably accurate (less than 3% and 5% error in predicting power and area, respectively). Finally, our dynamic refinement is able to find the optimal solution quickly (less than 10 iterations).

ACKNOWLEDGMENTS

This work was partially supported by the grant from National Science Foundation (CCF-1908131).

REFERENCES

[1] Hasini Witharana, Sahan Sanjaya, and Prabhat Mishra. Dynamic refinement of hardware assertion checkers. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2023.

[2] Hasini Witharana, Yangdi Lyu, Subodha Charles, and Prabhat Mishra. A survey on assertion-based hardware verification. *ACM Computing Surveys, 54 (11)*, 2022.

[3] Prabhat Mishra et al. Post-silicon validation in the soc era: A tutorial introduction. *IEEE Design & Test, 34(3)*, 2017.

[4] Kamran Rahmani, Sandip Ray, and Prabhat Mishra. Postsilicon trace signal selection using machine learning techniques. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(2):570–580, 2016.

[5] Harry Foster. Wilson research group functional verification study 2020.

[6] Hasini Witharana, Aruna Jayasena, Andrew Whigham, and Prabhat Mishra. Automated generation of security assertions for rtl models. *ACM Journal on Emerging Technologies in Computing Systems*, 19(1):1–27, 2023.

[7] Hasini Witharana, Yangdi Lyu, and Prabhat Mishra. Directed test generation for activation of security assertions in rtl models. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 26(4):1–28, 2021.
[8] Marc Boule, Jean-Samuel Chenard, and Zeljko Zilic. Assertion checkers in verification, silicon debug and in-field diagnosis. In *8th International Symposium on Quality Electronic Design (ISQED'07)*, pages 613–620. IEEE, 2007.
[9] Samuele Germiniani and Graziano Pravadelli. Harm: a hint-based assertion miner. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(11):4277–4288, 2022.
[10] Shobha Vasudevan et al. Goldmine: Automatic assertion generation using data mining and static analysis. In *DATE*, 2010.
[11] Samuel Hertz, David Sheridan, and Shobha Vasudevan. Mining hardware assertions with guidance from static analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(6):952–965, 2013.
[12] Leonard A Breslow and David W Aha. Simplifying decision trees: A survey. *The Knowledge Engineering Review*, 12(1):1–40, 1997.
[13] Kenneth L McMillan. The smv system. In *Symbolic Model Checking*, pages 61–85. Springer, 1993.
[14] Alessandro Danese, Nicolò Dalla Riva, and Graziano Pravadelli. A-team: Automatic template-based assertion miner. In *Proceedings of the 54th Annual Design Automation Conference 2017*, pages 1–6, 2017.
[15] Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499, 1994.
[16] Debjit Pal, Spencer Offenberger, and Shobha Vasudevan. Assertion ranking using rtl source code analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(8):1711–1724, 2019.
[17] Ranganathan Hariharan, Tara Ghasempouri, Behrad Niazmand, and Jaan Raik. From rtl liveness assertions to cost-effective hardware checkers. In *2018 Conference on Design of Circuits and Integrated Systems (DCIS)*, pages 1–6. IEEE, 2018.
[18] Tara Ghasempouri and Graziano Pravadelli. On the estimation of assertion interestingness. In *2015 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 325–330. IEEE, 2015.
[19] Mohammad Reza Heidari Iman, Jaan Raik, Gert Jervan, and Tara Ghasempouri. Immizer: An innovative cost-effective method for minimizing assertion sets. In *2022 25th Euromicro Conference on Digital System Design (DSD)*, pages 671–678. IEEE, 2022.
[20] Pouya Taatizadeh and Nicola Nicolici. Emulation infrastructure for the evaluation of hardware assertions for post-silicon validation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(6):1866–1880, 2017.
[21] Yusuke Kimura et al. Signal selection methods for efficient multi-target correction. In *ISCAS*, 2019.
[22] Farimah Farahmandi et al. Cost-effective analysis of post-silicon functional coverage events. In *DATE*. IEEE, 2017.
[23] Mohammad Hossein Neishaburi and Zeljko Zilic. An infrastructure for debug using clusters of assertion-checkers. *Microelectronics Reliability*, 52(11):2781–2798, 2012.
[24] Ming Gao and Kwang-Ting Cheng. A case study of time-multiplexed assertion checking for post-silicon debugging. In *HLDVT*, 2010.
[25] Tommy Tracy II, Lucas M Tabajara, Moshe Vardi, and Kevin Skadron. Runtime verification on fpgas with ltlf specifications. volume 1, pages 36–46. TU Wien Academic Press, 2020.
[26] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: an efficient data clustering method for very large databases. *ACM sigmod record*, 25(2):103–114, 1996.
[27] LowRISC/ibex. https://github.com/lowRISC/ibex.