Dynamic Refinement of Hardware Assertion Checkers

Hasini Witharana, Sahan Sanjaya and Prabhat Mishra
University of Florida, Gainesville, Florida, USA

***Abstract*—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 the 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. Prior efforts considered synthesis of a small set of checkers based on design constraints. However, these design constraints can change dynamically during the device lifetime due to changes in use-case scenarios as well as input variations. In this paper, we explore dynamic refinement of hardware checkers based on changing design constraints. Specifically, we propose a cost-based assertion selection framework that utilizes non-linear optimization as well as machine learning. Experimental results demonstrate that our machine learning model can accurately predict area (less than 5% error) and power consumption (less than 3% error) of hardware checkers at runtime. This accurate prediction enables close-to-optimal dynamic refinement of checkers based on design constraints.**

I. 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 [1]. Therefore, post-silicon validation is an essential step in SoC design methodology. One of the biggest challenges in post-silicon validation is the lack of observability of internal states. Typically, a small trace buffer is used to trace few hundred signals (out of millions of signals) during runtime [2]. A prominent avenue to improve post-silicon observability is to use hardware checkers (assertions). According to the 2020 Wilson research study [3], around 75% of ASIC design and 50% of FPGA design projects use assertion-based validation. However, assertions also introduce hardware overhead. Therefore, it is not practical to synthesize thousands or millions of pre-silicon assertions to post-silicon checkers.

There are early efforts [4], [5] to select the most beneficial set of assertions as hardware checkers based on area, power, and performance constraints. The selected assertions may not be beneficial since the design constraints can change dynamically during the device lifetime due to changes in use-case scenarios as well as input variations. For example, mobile phone usage pattern can drastically change between two users. Even for the same user, the usage of the phone varies during the different time periods of a day. In other words, different use-case scenarios and input variations can lead to dynamic changes in power and performance. Moreover, the dynamic changes 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, it would be beneficial to dynamically refine the checkers to satisfy both dynamically changing circumstances and runtime checking objectives.

This work was partially supported by the NSF grant CCF-1908131.

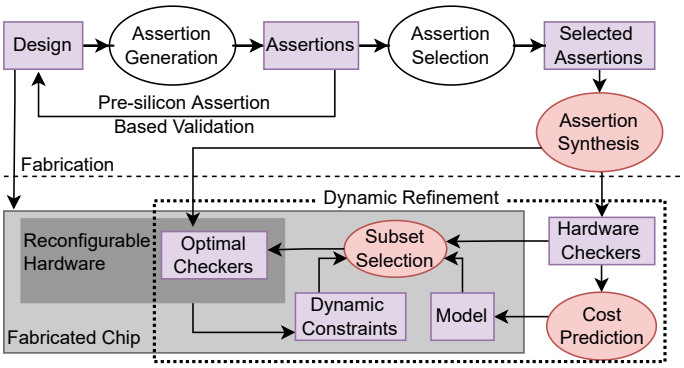


Fig. 1: Assertion-based validation framework. The dotted box (dynamic refinement) shows our proposed contributions.

Figure 1 shows a brief overview of our proposed framework in the context of assertion-based validation. During pre-silicon validation, assertions are generated for a given design [6], [7], [8]. Due to design overhead constraints, assertion selection [4], [5] can be used to identify the most profitable assertions. The selected assertions are synthesised as hardware checkers. The hardware checkers are used for our dynamic refinement framework. A regression model is trained to predict the cost (power and area) of synthesizing a set of checkers. When the design constraints change dynamically, the subset selection uses the regression model to select the optimal subset of checkers that satisfies the design constraints at that time. The selected checkers are synthesised in reconfigurable hardware. This paper makes the following major contributions:

- Formulates the dynamic refinement problem as a cost-based non-linear optimization problem.
- Uses regression based machine learning techniques to perform cost prediction for hardware checkers.
- Solves the non-linear optimization problem using gradient decent with simulated annealing.
- Demonstrates close-to-optimal dynamic refinement of hardware checkers.

This paper is organized as follows. Section II surveys related efforts. Section III presents the problem formulation. Section IV describes our proposed framework. Section V presents experimental results. Section VI concludes the paper.

II. Related Work

Pre-silicon assertions can be utilized during post-silicon debug by synthesizing them as hardware checkers [1]. A major challenge in assertion selection is to determine which assertions should be added to the design as hardware checkers. Profitable checker selection can be conducted using static synthesis with different ranking algorithms [9], [10], [11], [12]. The number of hardware checkers can be reduced [13], [4] by utilizing the existing debug infrastructure (trace buffer). Another promising alternative for cost effective hardware checkers is the dynamic synthesis of checkers using FPGA [14]. In this work, the hardware checkers are included in a re-configurable embedded block (FPGA) in a time-multiplexed manner. This approach enables to add a large number of checkers with a low area overhead. To the best of our knowledge, our approach is the first attempt to dynamically refine hardware checkers based on changing design constraints.

III. Problem Formulation

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:

$$\underset{S}{\text{minimize}}(\mathcal{F}(S))$$
$$\mathcal{P}(S) \leq P, \quad \mathcal{A}(S) \leq A$$

where S is a subset of checkers and $\mathcal{P}$ and $\mathcal{A}$ 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 checker) 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 $\mathcal{F}, \mathcal{P}$ and $\mathcal{A}$ given that there are 2^N possible inputs, where N 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^N 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 IV-A 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 satisfies 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 IV-B describes how our approach solves the optimization problem.

IV. Dynamic Refinement of Hardware Checkers

Figure 2 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.

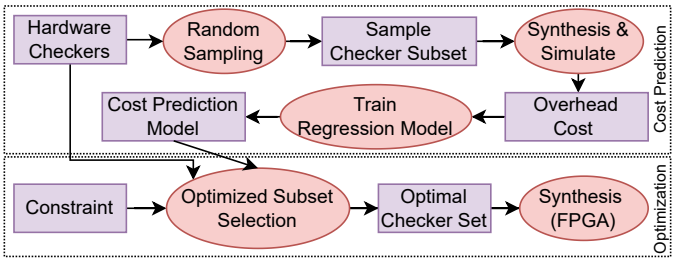


Fig. 2: Overview of our dynamic refinement scheme

A. Cost Prediction

In this section, we address the problem of estimating the functions $\mathcal{F}, \mathcal{P}$ and $\mathcal{A}$ that appear in the optimization problem. Typically, the function $\mathcal{F}$ is a simple cost model depending on the functions $\mathcal{P}(S)$ (power) and $\mathcal{A}(S)$ (area) 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 $S_1, \ldots, S_k$ and power consumption estimates $p_1 = \mathcal{P}(S_1), \ldots, p_k = \mathcal{P}(S_k)$ find a good approximation of the function $\mathcal{P}(S)$. The same solution can be used for $\mathcal{A}$ 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 $\mathbb{R}^k \longrightarrow \mathbb{R}$. 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 N that contains only 0.0 or 1.0 entries.

As shown in Figure 2, random sampling is conducted on hardware checkers to get different sample subsets. These sample subsets are synthesized and simulated to get the overhead cost with respect to power and area. The overhead costs are used to train the regression model. Once a learning model is built, it can be used to predict $\mathcal{F}(S)$ simply by encoding the input S using the same $0.0,\ 0.1$ mapping and using the predictor for the estimate.

B. Optimization

In order to solve the non-linear optimization problem (formulated in Section III), we use non-linear optimization techniques, including gradient descent and simulated annealing.

Algorithm 1: Gradient Descent

Data: Estimators for $\mathcal{F}(S)$ & $\mathcal{C}(S)$
Result: Locally optimal solution S

```
1  Find starting point;
2  repeat
3  |  select random S
4  until C(S) is true;
5  Done ← False;
6  while ¬Done do
7  |  Done ← GD(S)
8  end
9  Function GD(S):
10 |  Keep track of the best solution;
11 |  S' ← S;
12 |  for i ← 1 to N do
13 |  |  Add or remove checker C_i to S;
14 |  |  S_i ← S ⊕ C_i;
15 |  |  if C(S_i) & F(S_i) < F(S') then
16 |  |  |  S' ← S_i
17 |  |  else
18 |  end
19 |  if S' ≠ S then
20 |  |  New better solution;
21 |  |  S ← S';
22 |  |  Done ← False;
23 |  else
24 |  |  Reached local minimum;
25 |  |  Done ← True;
26 |  end
27 return Done
28
```

The gradient descent algorithm used in our framework is presented in Algorithm 1. Inputs of the algorithm are $\mathcal{F}(S)$ and $\mathcal{C}(S)$. Function $\mathcal{F}(S)$ is retrieved using the cost prediction model described in Section IV-A. Function $\mathcal{C}(S)$ represents a Boolean function which 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 satisfy $\mathcal{C}(S)$. Gradient descent method starts with a random initial point (line 1 - 3). Function GD is repeated until the local optimal solution is found (line 6 - 8). Function GD (line 9 - 27) presents the gradient descend method for finding the optimal solution. This method takes steps that stay within the feasible region (i.e., satisfy the constraints) and decrease the cost. Eventually, points where this is no longer possible (i.e., local minimums) are reached. The step function of the gradient descent method is defined as the addition or removal of each checker C_i to the current best set S at every step (line 12 - 18). Here, N means the number of hardware checkers (line 12). The optimal solution S is selected based on the constraint satisfaction and the minimal cost prediction (line 15 - 17). The termination condition checks whether all the neighbors of S are worse. The neighbors are the sets that differ by at most one element. If all neighbors are worse, S is considered as a local minimum (line 19 - 26). The algorithm guarantees feasible solutions since the condition $\mathcal{C}(S)$ is checked for both the initial point and each S_i candidate. By considering multiple random restarts for Algorithm 1 and keeping track of the best solution, optimized set of checkers can be found for a given set of constraints.

Algorithm 2: Gradient Descent + Simulated Annealing

Data: Estimators for $\mathcal{F}(S)$ & $\mathcal{C}(S)$, Max, Prob
Result: Locally optimal solution S

```
1  Find starting point;
2  repeat
3  |  select random S
4  until C(S) is true;
5  Probability of a random step;
6  p ← 1/2;
7  Annealing loop; probability gets halved;
8  while p > Prob do
9  |  Done ← False;
10 |  Steps took since last change in p;
11 |  Steps ← 0;
12 |  while ¬ Done & Steps < Max do
13 |  |  Steps ← Steps + 1;
14 |  |  Coin ← FlipCoin(p);
15 |  |  if Coin == Head then
16 |  |  |  Take random step;
17 |  |  |  i ← random(1,n);
18 |  |  |  S ← S ⊕ C_i;
19 |  |  else
20 |  |  |  Gradient descent step;
21 |  |  |  Done ← GD(S)
22 |  |  end
23 |  end
24 |  p ← p/2;
25 end
```

In general, when the optimization problem is relatively simple (i.e., it has a small number of local minimums), gradient descent methods perform fairly well. We do not expect this to be true for our optimization problem since the interaction between the hardware checkers is likely to be complicated. In such situations, simulated annealing methods are preferred. The basic idea is to modify the gradient descent strategy by adding random feasible steps that allow local minimums to be escaped. 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. Algorithm 2 depicts this more complicated process.

Algorithm 2 presents a gradient descent search of the subset space with annealing. The algorithm begins at a random initial feasible subset (line 2 - 4). First the probability of random step is given value 0.5 (line 6). Then this probability is halved through the annealing loop (line 8 - 25). The loop is conducted for 'Max' steps by checking whether the probability is greater

than 'Prob'. For each iteration, 'FlipCoin' is conducted to determine whether we will move to the best feasible neighboring subset (line 19 - 22) or will move to a random feasible neighboring subset (line 15 - 18). 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 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, gradient descent function GD in Algorithm 1 is used (line 21). The probability of a random move decreases by a factor of 0.5 every 'Max' steps (line 24). 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.

V. Experiments

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

A. Experimental Setup

For the experimental evaluation, we have selected benchmarks from TrustHub [15], OpenCores [16] and RISC-V CPU core [17] as shown in Table I. The first column of the table shows the benchmarks. The second and third column show the number of LUTs in the design and the number of assertions selected as hardware checkers for synthesis, respectively. The last column shows the number of samples used for the cost prediction model. The assertion generation was conducted manually as well as using Goldmine [6]. 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, of both the concurrent and immediate form, 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.

TABLE I: Benchmarks

Benchmark	# LUT	# Checkers	Sample Size
D-Cache	146	15	1000
Ibex Decoder	152	20	1000
Ibex Controller	159	10	300
PCI	222	12	500
Ibex ID-Stage	425	20	1000
AES	1765	10	300

B. 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 3 presents the power consumption (in watts) for different number of checkers for PCI 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 12 individual checkers is 0.087 watts. However, when we get the 12 checkers together, the power consumption is 0.018 watts. This shows that out 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.

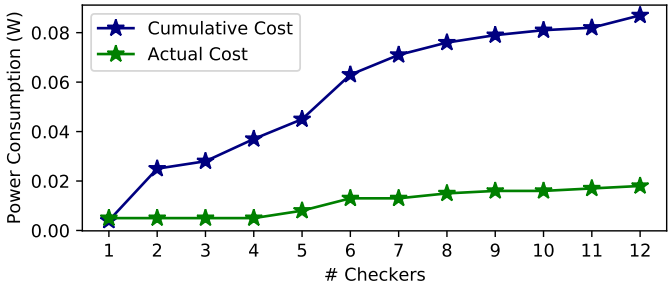
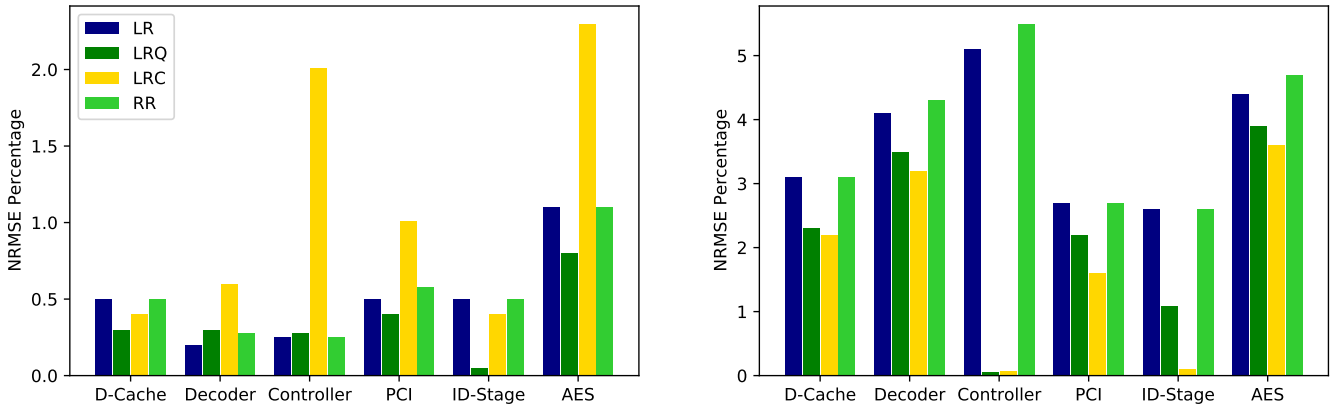


Fig. 3: Power consumption (W) for checkers in PCI

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 PCI, a sample of 500 subsets was collected, covering 12.21% of all possible subsets. Similarly, in case of D-Cache, 1000 subsets were collected, covering 3.05% 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



(a) Model accuracy for power consumption prediction

(b) Model accuracy for LUT prediction

Fig. 4: Model accuracy for different regression models

TABLE II: Dynamic refinement results for PCI with different iterations

LUT	Power(W)	Iterations = 10	Iterations = 20	Iterations = 30	Iterations = 40	Iterations = 50
5	0.125	"001000010100" (3) P=0.124, LUT=3.61	"000000011100" (3) P=0.122, LUT=4.85	"000000011100" (3) P=0.122, LUT=4.85	"000000011100" (3) P=0.122, LUT=4.85	"000000011100" (3) P=0.122, LUT=4.85
5	0.155	"001000010110" (4) P=0.127, LUT=4.75	"101000010100" (4) P=0.127, LUT=4.8	"001000010110" (4) P=0.127, LUT=4.75	"101000010100" (4) P=0.127, LUT=4.8	"101000010100" (4) P=0.127, LUT=4.8
25	0.135	"100110011111" (8) P=0.134, LUT=16.9	"001110011111" (8) P=0.134, LUT=16.9	"101110011101" (8) P=0.133, LUT=16.8	"101110011101" (8) P=0.133, LUT=16.8	"001110011111" (8) P=0.134, LUT=16.9
45	0.155	"111111011111" (11) P=0.144, LUT=27.89	"111111011111" (11) P=0.144, LUT=27.89	"111111011111" (11) P=0.144, LUT=27.89	"111111011111" (11) P=0.144, LUT=27.89	"111111011111" (11) P=0.144, LUT=27.89

validation, with normalized root mean squared error (NRMSE) as the performance metric. The performance evaluation for all the 4 models for each benchmark with respect to power and LUT consumption is shown in Figure 4. For all the designs, the four models achieved less than 3% error predicting the power consumption and less than 5% error predicting LUT.

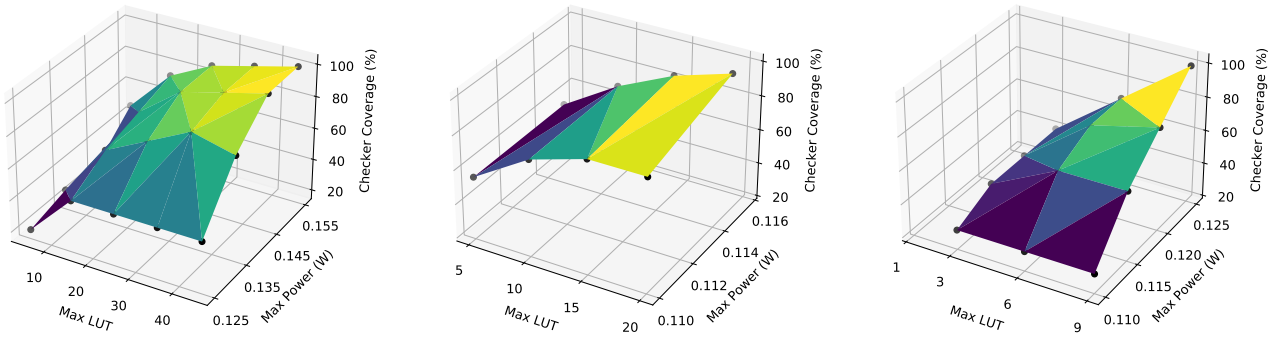
The model with the best performance on the test set for each metric and design pair (Figure 4) was then utilized during the subset selection algorithm in dynamic refinement to predict the LUT and power overhead for potential selected hardware checker subsets. By estimating whether a checker subset satisfied the applied constraints, the subset selection algorithm produced results without the processing bottleneck of HDL synthesis for each subset and its neighbors.

C. Dynamic Refinement Results

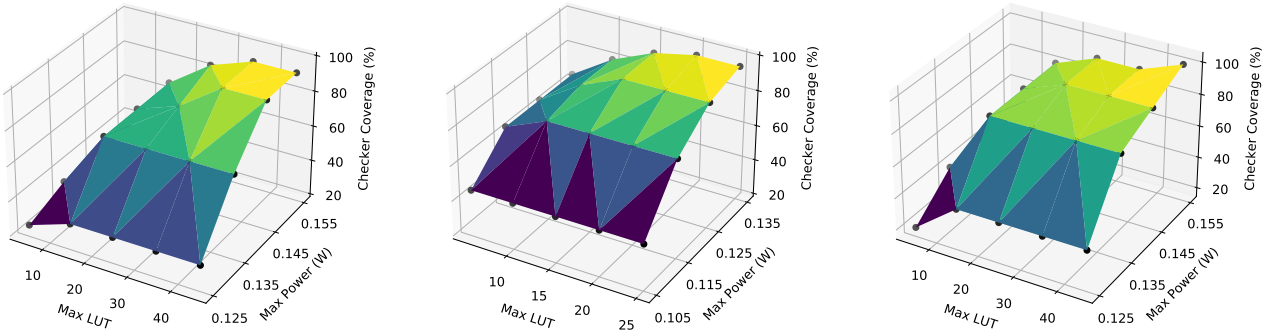
The subset selection algorithm was run for all designs with a variety of parameters, including LUT constraints, power constraints, and number of iterations (by changing the 'prob' value). Figure 5 shows the dynamic refinement of hardware checkers for each benchmark with different constraints. For each 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 5 shows the optimal selection of hardware checkers for each constraints pair chosen from the results from different iterations.

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.

The results of running the dynamic refinement algorithm for PCI benchmark with increasing iterations (10 to 50) are shown in Table II. 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 11 (91.67%) with optimal solution even from



(a) D-Cache (N=15) (b) Ibex Decoder (N=20) (c) Ibex Controller (N=10)



(d) PCI (N=12) (e) Ibex ID-Stage (N=20) (f) AES (N=10)

Fig. 5: Checker subset selection for changing requirements

10 iterations for LUT=45 and Power = 0.155 W on the PCI design. However, for some constraints increasing the number of iterations helped to achieve the optimal solution (LUT=5 and Power=0.155).

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.

VI. Conclusion

Post-silicon validation and in-field debug relies on observability infrastructure such as trace buffers. Hardware checkers can improve the observability for debugging functional as well as non-function (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 not useful under changing workloads and input variations. We presented a framework to dynamically refine hardware checkers for changing design constraints. 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. Experimental evaluation demonstrated the effectiveness of our framework.

References

[1] H. Witharana, Y. Lyu, S. Charles, and P. Mishra, "A survey on assertion-based hardware verification," *ACM Computing Surveys, 54 (11)*, 2022.
[2] P. Mishra *et al.*, "Post-silicon validation in the soc era: A tutorial introduction," *IEEE Design & Test, 34(3)*, 2017.
[3] H. Foster, "Wilson research group functional verification study 2020."
[4] F. Farahmandi *et al.*, "Cost-effective analysis of post-silicon functional coverage events," in *DATE*. IEEE, 2017.
[5] P. Taatizadeh and N. Nicolici, "Automated selection of assertions for bit-flip detection during post-silicon validation," *TCAD*, 2016.
[6] S. Vasudevan *et al.*, "Goldmine: Automatic assertion generation using data mining and static analysis," in *DATE*, 2010.
[7] H. Witharana *et al.*, "Directed test generation for activation of security assertions in rtl models," *ACM TODAES*, vol. 26, no. 4, 2021.
[8] ——, "Automated generation of security assertions for RTL models," *ACM Journal on Emerging Technologies in Computing Systems*, 2022.
[9] M. Eslami *et al.*, "Reusing verification assertions as security checkers for hardware trojan detection," *arXiv preprint arXiv:2201.01130*, 2022.
[10] R. Hariharan *et al.*, "From rtl liveness assertions to cost-effective hardware checkers," in *DCIS*, 2018.
[11] A. Adir *et al.*, "Leveraging pre-silicon verification resources for the post-silicon validation of the ibm power7 processor," in *DAC*, 2011.
[12] P. Taatizadeh and N. Nicolici, "Emulation infrastructure for the evaluation of hardware assertions for post-silicon validation," *VLSI*, 2017.
[13] Y. Kimura *et al.*, "Signal selection methods for efficient multi-target correction," in *ISCAS*, 2019.
[14] M. Gao and K.-T. Cheng, "A case study of time-multiplexed assertion checking for post-silicon debugging," in *HLDVT*, 2010.
[15] "Trusthub," https://www.trust-hub.org/.
[16] OpenCores, https://www.opencores.org/, 2020.
[17] "LowRISC/ibex," https://github.com/lowRISC/ibex.