Machine Learning-Guided Stimulus Generation for Functional Verification

Saumil Gogri, Jiang Hu, Aakash Tyagi, Mike Quinn, Swati Ramachandran, Fazia Batool, and Amrutha Jagadeesh

Department of Electrical and Computer Engineering, Texas A&M University, College Station, TX 77843
Email: gogrisaumil@gmail.com Phone: 979-481-7577

Abstract

As chip complexity continues to grow, simulation-based functional verification is becoming a bottleneck for design turnaround time. This problem can potentially be mitigated by machine learning-guided (ML-guided) stimulus generation that attains verification coverage with considerably reduced simulations. Although there can be several different ways for realizing such generation, related previous works mostly restrict their attention to only one or two and there lacks a systematic study. Additionally, a large portion of the previous work was produced in the early years of ML and thus the recent progress in ML technology is largely missing in this field of study. In our work, we study ML-guided stimulus generation at multiple hierarchies, along with the development of new techniques and exploitation of leading-edge ML technology. Experimental results indicate that test-level optimization, which is coarse-grained, is not always effective. The cases difficult for test-level optimization can be well solved by our transaction-level optimization, which is fine-grained. Our ternary classification technique is shown to be superior to the conventional binary classification. Overall, our techniques lead to about 70% reduction in verification time (total CPU-compute time) even when model training and data processing time are counted.

I. Introduction

Introduction: RTL simulation-based functional verification is a critical yet enormously time-consuming step for many digital IC designs. In the past, Machine Learning (ML) techniques have been explored to alleviate the redundancy in random simulations in an attempt to reduce verification time. However, much of the work [1], [2], [3], [4] was carried out only before the major wave progress of ML technology. A more recent ML approach [5] is restricted to toggle coverage and another work [6] applies neural network for verification stimulus pruning. Despite a successful demonstration of the effectiveness of ML techniques, there are still several issues that remain to be addressed.

- **Coarse-grained or fine-grained ML application?** Stimulus can be organized into a multi-level hierarchy with different granularity levels. The previous works mostly dived into one level without mentioning the other levels. Therefore, it is unclear what is the best granularity level for ML applications.
- **Stimulus pruning or constructive stimulus generation?** The ML guidance can be provided in two different ways: either to prune out generated stimulus or to generate new stimulus constructively. Again, most previous works took one way without comparing with the other.
- **FSM (Finite State Machine) versus non-FSM.** Although FSM is a prevalent circuit style, previous ML-based work rarely dedicated customized solutions for it, and often treated FSM and non-FSM alike.
- **Which ML engine to use?** There are quite a few options for different ML engines - neural network, SVM (Support Vector Machine), linear regression, random forest, etc. Most of the previous works were restricted to a single engine without comparing with others. As such, it is not clear which engine fits the best for guiding stimulus generation. Moreover, the previous works were mostly conducted before 2012. As a result, some recently popular engines, such as random forest and LSTM (Long Short Term Memory), have not received sufficient attention.

Partly due to these issues, ML-guided stimulus generation has not been widely adopted in the industry. The goal of our work is to address these issues in a more thorough study and develop new techniques for further improvement. Our work offers the following contributions.

1) Both coarse-grained test-level and fine-grained transaction-level optimization are studied and compared. While the coarse-grained approach is effective in some cases, there are cases where our fine-grained techniques succeed and coarse-grained optimization fails.
2) For coarse-grained test-level optimization, a logistic regression-based ternary classification technique is suggested and shown to be superior to conventional binary classification.
3) Both stimulus pruning and constructive stimulus generation techniques are developed and compared. The constructive approach slightly outperforms pruning-based methods.
4) A graph-based constructive transaction generation technique is developed for FSM designs.
5) For fine-grained stimulus optimization on non-FSM designs, the history effect is studied.
6) Multiple ML engines, especially the recently popular deep neural network and random forest, are evaluated and shown to outperform the traditional SVM.

7) Experimental results show that ML can reduce overall verification time by 70% even when the model training time is considered. For some cases, a fine-grained approach takes 50% less verification time than a coarse-grained method.

The proposed methodology is a completely automated process and thus can avoid costly manual engineering required by directed random tests.

II. Related Work

Recent years have seen an uptick of efforts in the exploration of ML in various aspects of functional verification. Stan Sokorac [5] proposed a methodology where toggle coverage metrics and ML algorithms are utilized to create tests with higher potential of exposing previously uncovered regions in the design. Unfortunately, toggle coverage is not a very effective metric to measure actual design functionality. In another work [7], clustering and neural network techniques are implemented to reduce the time taken to collect and evaluate coverage. However, this work does not take stimulus optimization into consideration. A method for neural network-based transaction-level optimization is introduced in [6]. However, its model training time is significantly greater than time saved for design verification.

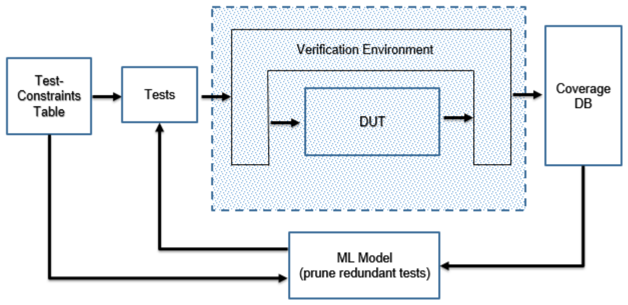
ML-based Coverage Directed Test Generation (CDTG) began in early 2000s, when a Bayesian network approach was introduced in [1]. A survey [4] on ML-based CDTG was published in 2012. It covered various ML techniques, such as genetic algorithms, Bayesian networks [1], Markov models [8], and data mining [9]. In [2], [3], the authors propose stimulus optimization through test pruning using support vector analysis (SVA). In [2], a graph-based kernel function is defined using design knowledge to calculate the similarity between pairs of test, which are then grouped using an unsupervised learning technique – clustering. Then only selected tests from each cluster are simulated. Test grouping based on similarity of coverage information is proposed in [3]. Here the toggle coverage is considered as the coverage metric and assembly level tests form the stimulus. The coverage prediction is done per instruction through indexing an instruction coverage table based on Hamming-distance calculation between the operand values of the instruction in the table.

Despite their contributions, the previous works mostly share the same issues described in Section I. First, the impact of different granularity levels for ML application has not been well studied. Second, it is not clear how constructive stimulus generation compares with stimulus pruning. Third, FSM and non-FSM designs have not been well differentiated. Fourth, their ML classifications are mostly binary, which is a disadvantage compared with our ternary classification approach. Last but not least, state-of-the-art ML engines, such as DNN (Deep Neural Network), random forest, and LSTM, have not been explored.

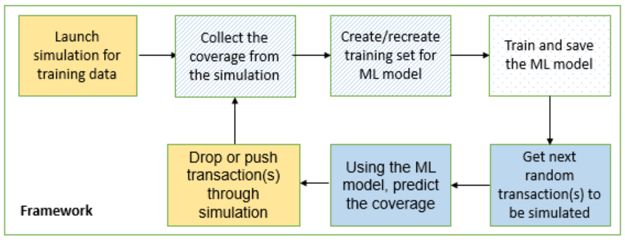
III. ML-Guided Stimulus Generation

The section describes our study on both test-level coarse-grained and transaction-level fine-grained stimulus optimization guided by machine learning, including newly proposed techniques.

A. Coarse-Grained Test Optimization



(a) Test-level stimulus optimization flow.


Launch simulation for training data
Collect the coverage from the simulation
Create/recreate training set for ML model
Train and save the ML model
Get next random transaction(s) to be simulated
Using the ML model, predict the coverage
Drop or push transaction(s) through simulation
Framework

(b) Transaction-level optimization flow

Fig. 1: ML-guided stimulus generation framework

Test-level stimulus can be controlled through test-constraints, which can help steer the stimulus towards a particular design functionality. Here, we introduce test-level optimization to predict functional coverage of each non-simulated test based on the test-constraint values and the knowledge of already simulated tests, during an ongoing volume regression. This projection is utilized to prune tests that yield redundant coverage and thus directing the stimulus towards the uncovered areas in the design. A ML-guided test-level optimization flow is illustrated in Figure 1a. The shaded box in the diagram is the existing testbench, which is unaltered.

The ML-guided verification flow is generally composed of two phases. In phase 1, conventional test generation and simulation are performed while an ML model is trained with the simulation results. In phase 2, the model is applied to prune out tests that do not improve coverage while the model continues to be trained.

Phase 1 begins with creating the test-constraints table. The software tool generates the table with random values, such that it forms legal stimulus for the design. For example, if only read-type transactions can be executed on instruction cache then random test-constraints generation will abide by it. Each entry in this table provides random values for all test-constraints required at run-time to simulate a random test. A volume regression is launched for simulating many such tests with fully randomized test-constraints. The simulation output data is collected in a coverage database.

Test-constraints and the coverage information from all simulated tests serve as the training data for the ML model. The test-constraints for each test form the input features to the ML model and coverage recorded on all the bins make the output labels for the prediction. The output labels are binary, i.e., it takes value 1 (bin covered) and 0 (bin uncovered). The ML model is trained and validated in the background during the ongoing regression until the training error falls below a certain threshold value-γ. The training error is calculated by taking the mean of the validation-set error and 5-fold CV (Cross Validation). The former is calculated by separating 20% of the training set as a validation set. The model is trained with the remaining training set and classification error is evaluated on the validation set. To calculate the 5-fold CV error, the training set is shuffled and randomly divided into 5 groups. For each unique group, it is held out as a validation set and the model is trained with the remaining data to find the classification error on it. This is repeated multiple times and the final error is calculated by taking the average of all errors. Lower validation-set error projects that the model performs well on unseen data while limiting 5-fold CV error will make sure the model is not biased towards the training data.

Phase 2 starts when the training error drops below γ and then the trained model is used for coverage prediction. Test-constraint values for succeeding non-simulated tests are fetched from the test-constraint table sequentially. Previous works mostly apply ML for *binary* classification to predict if a location in functional space will be covered by a test. There could be cases that are not obvious to the model and easily result in classification error.

In our daily life, we usually take a closer look at difficult cases instead of rushing into conclusion. Likewise, we propose a ternary classification among three cases: obviously covered, obviously not covered and not obvious if covered or not. To facilitate the ternary classification, we first apply logistic regression, which outputs probability instead of binary results. For example, logistic regression for weather forecast predicts the probability of rain. Such logistic regression can be obtained from many ML engines including DNN and random forest. For a logistic regression result p, we carry out a ternary classification on every bin according to two thresholds $0 < \beta < \alpha < 1$. If $p \geq \alpha$, the corresponding bin is classified as decided-1, which means the bin will be hit by the test. If $p \leq \beta$, the bin is treated as decided-0, which implies that the bin will not be hit by the test. If $\beta < p < \alpha$, the coverage bin falls in the undecided bucket, i.e., the model is uncertain about the coverage value. The proposed ternary classification can be transformed to binary classification by setting values of α and β to 0.5, i.e., a bin can be classified to either decided-1 or decided-0.

The test pruning decision is undertaken based on this ternary classification of all the bins corresponding to the coverage metric. If a test leads to a high percentage of bins with decided-1 or -0 coverage and those decided-1 bins have already been covered, simulation is not performed for this test as it is not likely to improve coverage. Contrarily, if the model predicts decided-1 on an uncovered bin for a test, it is simulated. Additionally, if the test involves a fair number of bins with undecided coverage, it is simulated. More bins with undecided coverage imply that the ML model faces difficulty in coverage prediction for the test. Thus, the test has higher odds of generating contrasting stimulus and hitting the coverage not covered by the training data. All additional test simulation results are utilized to re-train the ML model for improving prediction accuracy over following non-simulated tests.

The test-level optimization framework encapsulates the existing testbench with no modifications inside it. Hence, it is a scalable approach and can be applied to any existing hardware verification testbench. The ternary classification of the coverage prediction value is the key idea for test pruning, which differentiates our approach from most previous works. In the experiment, we evaluated multiple different ML engines, including DNN, random forest in addition to SVM, which is employed in the previous work [3].

B. *Fine-Grained Transaction Optimization*

The test-constraints impart low controllability on the type of transaction generation in any sequence of the test. For example, it is difficult for a test-level constraint to deterministically generate a sequence of read followed by write and further followed by read transaction to meet a certain coverage goal. Moreover, not all the transactions, simulated as part of the test targeted for coverage closure, help improve coverage. Therefore, we propose and study fine-grained transaction-level stimulus optimization.

Although transaction-level stimulus optimization shares a very similar goal and strategy as that of test-level, which is to build an ML model to recognize the correlation between each transaction and coverage such that transactions leading to redundant coverage are avoided in simulation, there is a key difference. That is, only transaction attributes alone are insufficient for obtaining an effective model, unlike test-level where test-constraints are adequate. We include transaction context or history into consideration. Further, we differentiate FSM (Finite State Machine) and non-FSM components with different coverage metrics and approaches.

The general flow of transaction-level optimization is depicted in Figure 1b. Like test-level optimization, it is also divided into two phases. Phase 1 is like conventional random simulation while an ML model is trained. In phase 2, the model is applied to improve coverage with fewer simulations while the model is continuously trained. For each of FSM and non-FSM approaches, we investigate two methods to utilize the coverage prediction for accelerated coverage closure, discussed as follow.

1) **Online transaction pruning**: This is an online approach where transactions are processed for coverage prediction in the middle of simulating a sequence. If some transactions are classified by the ML model as not helpful in improving coverage, they are pruned out of environment without being simulated.
2) **Offline ML-directed sequence generation**: Based on ML predictions, a sequence composed of transactions improving the overall coverage is generated offline for simulation. This is very similar to manually written directed test, but here the ML model is allowed to craft the sequence directed for coverage closure.

1) FSM Transition Coverage: For FSM designs, an important metric is state transition coverage, which is the main focus of our study, although our methods can be easily extended for state coverage. A state transition depends on the current state and the type of new transaction characterized by its attributes. Thus, the ML model takes transaction attributes and current state as its input features and outputs the prediction of the next state, as shown in Figure 2. In phase 1, random tests are simulated and the results are applied to train the ML model. Once the model is well trained (i.e. training error $< \gamma$), phase 2 starts and the model is employed for covering remaining holes with either transaction pruning or directed sequence generation.

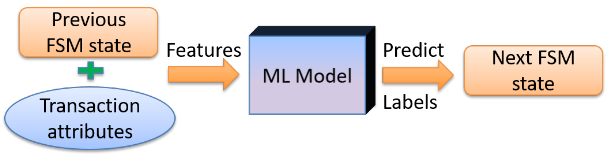


Fig. 2: ML setup for FSM type metric

In online transaction pruning, the application of the ML model is relatively straightforward. Before each randomly generated transaction of a sequence is fed to the design for simulation, the state transition is examined by the ML model to tell if it helps to improve the transition coverage. Only those helpful set of transactions are simulated while the others are pruned out.

We propose a graph-based offline sequence generation technique for FSM transition coverage. A TA (Transaction Attribute) transition graph $G_{TA}(V, E)$ is constructed with each node $s \in V$ in the graph corresponding to an FSM state and each directed edge $(s_i, s_k) \in E$ signifying a state transition. The nodes and edges are pretty much the same as those of state transition diagram. The key difference is that each edge in G_{TA} is associated with a transaction attribute vector $\mathbf{a}$. Usually, the FSM state space in a design is quite large and the dependence of the state transitions on transaction attributes is not known *a priori*. Therefore, the TA transition graph has to be constructed during transaction simulations. Suppose current state is s_i, by simulating a transaction with attribute $\mathbf{a}_j$, the next state s_{i+1} is reached. Then, an edge (s_i, s_{i+1}) with attribute $\mathbf{a}_j$ is added to the TA transition graph.

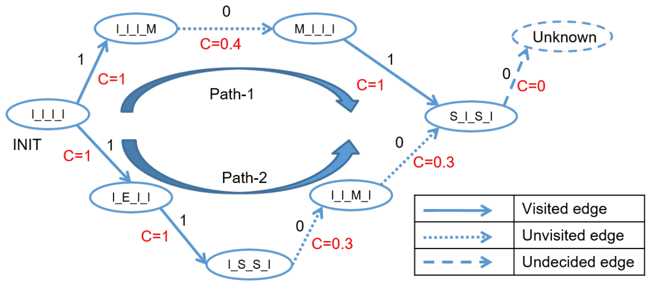


Fig. 3: Coverage Closure using the Shortest Path Algorithm on TA-Predicted Graph

After phase 1 in our simulation flow, a machine learning model is trained to predict the next state, for a given state and a given transaction attribute vector. Next, a TA transition graph G_{TA} is built in two stages.

- Stage 1 is to form a TA-trained graph, which is a subgraph of G_{TA}, according to the completed simulation data so far, i.e., the training data to the ML model. In the TA-trained graph, every edge corresponds to a state transition that has already been simulated in phase 1, and is labeled as *visited*. States and transitions that have not appeared in the simulations are not in the TA-trained graph.
- In stage 2, the TA-trained graph is augmented with new nodes and edges according to ML prediction to obtain a TA-predicted graph.

Please note that the ML model may lead to two cases of invalid predictions. In case 1, the ML predicts that no next state, including the current state, is reached. For example, the prediction for a 4-state system with states $\{s_3, s_2, s_1, s_0\}$ is represented by 4-bit binary vector, such as $(0, 0, 1, 0)$ indicating the next state s_1. Then, prediction result $(0, 0, 0, 0)$ is the first case of invalid prediction. In case 2, the ML model predicts multiple next states, e.g., $(0, 0, 1, 1)$. In the TA-predicted graph, the edges associated with invalid predictions are labeled with *undecided* and one end of such edge is designated as *unknown* node. Edges for valid predictions are labeled with *unvisited*.

One example of TA-predicted graph is shown in Figure 3. Cost is further assigned with each edge in the TA-predicted graph. Each *visited* edge has cost $C = 1$ while each *undecided* edge has cost $C = 0$. The cost of each *unvisited* edge is 1 minus the predicted probability p of the corresponding transition according to the ML model. One can see that a sequence of transactions corresponds to a path in the TA-predicted graph. In order to reach unvisited transitions in a new sequence, we find a shortest path on the graph. Since the ML model fails to predict a valid transition for those *undecided* edges, simulations are needed for these edges and 0 cost would encourage to include them in the new transaction sequence. If the ML predicts an unvisited transition with high probability p, the $C = 1 - p$ edge cost is a low value for including this transition in the new sequence. In phase 2, the following steps are repeated till the coverage goal is reached.

1) Construct (or update) the machine learning model according to the simulated data so far.
2) The TA-predicted graph is constructed (or updated) according to the simulated data so far.
3) Take the current state in simulation as source node s_{source}.
4) Find target node s_{target}. If there is any *unknown* node, an *unknown* node is taken as the target node. Otherwise, take a node associated with an *unvisited* edge as target node s_{target}.
5) Find a shortest path π from s_{source} to s_{target} on the TA-predicted graph using the Dijkstra's algorithm.
6) Simulate the sequence of transactions corresponding to π.

In the example of Figure 3, the design is in INIT state. To reach the source node $\{S, I, S, I\}$ of the undecided edge, i.e., the source of the directed edge to be visited, there are two possible paths. If weights are assigned as shown, based on the prediction confidence of the model, the recommended path with the lowest sum of weights is path-1 with a total of 2.4 against 2.6 for path-2. Contrarily, assigning the same weight (0) to all coverage holes will result in path-2 as the recommended path with two unvisited edges.

For most instances, due to the multi-core system, the model learns the behavior from per core activity & incoming transaction attribute values and predicts the next state with high confidence. For example, the model learns the behavior from a sequence of read-write-read transactions on core 0 and if a similar stimulus pattern is seen on core 1, it can predict the design response correctly, provided the initial state of the design are the same.

2) Non-FSM event coverage: In certain designs without FSM, it is of interest to verify the DUT for some interesting events like a buffer full or cache hit. The occurrence of such events often depends on a sequence of transactions simulated in a specific order. It is very difficult, if not impossible, for test-level constraints or knobs to deterministically generate such ordered sequences. Therefore, we suggest making use of machine learning for transaction-level control on such sequences. As the coverage metric is non-FSM, there are no circuit states for tracking the history dependence in transactions. Hence, history dependence needs to be explicitly captured by the ML model. We adopt a recurrent neural network (RNN) model, where the outputs are fed back to the network input. The relationship between RNN and DNN is similar to that between sequential logic and combinational logic. RNN is effective for handling time series and history dependence, where the transactions later in the sequence would have a higher impact on the prediction. In particular, we will adopt LSTM (Long Short Term Memory) [10], which is the state-of-the-art RNN technique. As shown in Figure 4, the input features for RNN model are the attributes of an ordered sequence of w transactions and the output label corresponds to the prediction whether the event of interest is observed or not. The value of w controls the depth of simulated transaction history considered for making coverage prediction on the intended metric.

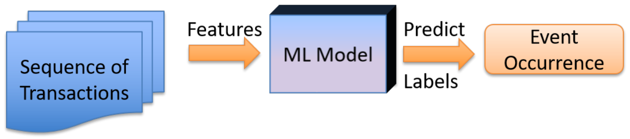


Fig. 4: ML setup for Non-FSM type metric

Same as the other methods introduced so far, the verification flow is composed of two phases, where random simulations are performed in phase 1 and ML training is performed at its end. In phase 2, we perform coverage closure based on prediction carried out by the trained model.

In online transaction pruning, the next transaction to be simulated along with $(w - 1)$ already simulated transactions in a test are examined by the RNN model. If the model predicts that they will trigger an event that has been covered, then the transaction is pruned out without being simulated. For any unknown or uncovered event, simulation is performed. If no single new transaction addition will improve event coverage according to the ML model, two new transactions are fetched from sequence generator and coverage prediction is performed on them plus $(w - 2)$ latest simulated transactions. New transactions are fetched from sequence generation unit one at a time until the prediction does not show redundant coverage.

In offline directed sequence generation, the software tool generates input feature corresponding to all *w* transactions randomly and loads into the ML model for coverage prediction, instead of the sequence generator in the testbench. If the coverage prediction results in an unknown or uncovered event, the set of feature values are translated to create a sequence of *w* transactions for simulation. Please note multiple different permutations of the same w transactions are individually evaluated by the ML model. In online pruning, by contrast, only one permutation can be evaluated by the ML model as the first $w-1$ transactions have already been simulated. Due to this difference, our offline method can examine more varieties of transaction sequences before simulation and potentially identify a sequence with greater improvement on verification coverage.

IV. Experiment Setup

The RTL design used in this study is a Quad-Core Cache design where the DUT is comprised of four L1 caches private to each core, a shared L2 cache, System Bus, Arbiter, and Memory. Each cache module is 4-way associative. The DUT implements functional aspects of MESI-cache coherency protocol and pseudo-LRU replacement policy. The testbench is fully UVM compliant with four drivers, one corresponding to each core and a scoreboard unit to verify the functional correctness of the design. The simulations are performed using the Cadence Incisive environment.

All the optimization frameworks are developed in Python; the libraries like Keras [11] and Scikit-Learn [12] are used to implement the four different ML models: DNN (Deep Neural Network), RF (Random Forest) [13], SVM (Support Vector Machine) and LSTM (Long Short Term Memory) [10], which is the state-of-the-art RNN technique.

A. Setup for Test-Level Optimization

A random test template in the testbench has 24 test-constraints (integers) like *core select, request type, access type, fix address (bit), sequential/parallel (bit),* and likewise. These are translated to 24 input features for the ML model. For coverage prediction, 8 coverage metrics in the design are considered with a total of 1738 bins, and each bin corresponds to an ML output label to predict. A few coverage metrics such as *core X request type, core X snoop, address X request types*, are simple and have a strong correlation with certain test constraints. For some other metrics like *combined snoop request*, only weak correlation with certain test constraints can be found. For complex coverage metrics like *MESI (FSM) transition* and *cache-hit X Address X core*, almost no such correlation exists and it is nearly impossible to deterministically reach such coverage by manipulating test-constraints.

Each coverage metric has a separate instance of ML model for the prediction, as the coverage for any two different metrics is mutually exclusive. First, we pick a coverage metric and utilize limited training data to determine the best ML model architecture and hyper-parameter configuration for the coverage prediction. The model is fine-tuned until the most reduced classification error is obtained. Then, for each of the remaining metrics, a proportionate structure is created. For example, the depth of the DNN structure is decided based on the ratio of output labels to input features (3 or 4 hidden layers), with the number of neurons in the middle hidden layers about twice the output labels. Each layer is followed by 20% dropout and activated with "ReLU", except for the last layer activated with "sigmoid" function. For RF, the depth of the decision trees and the forest size are kept proportionate to the number of the output labels. The ternary classification is applied to classify for decided-covered (1), decided-uncovered (0) or undecided bin. The values α=0.9 and β=0.1 were determined via experiments. We also implemented binary classification, which is equivalent to $\alpha = \beta = 0.5$.

During volume regressions, new random tests are simulated (in batches of 10 tests) and the ML model is trained in the background using the collective data from all simulated tests until the training error fall below $\gamma = 10\%$. In our case, the initial training goal is attained after training the model with data with first 50 random tests. In phase 2, the ML model is additionally trained after every 10 new simulated tests.

B. Setup for Transaction-Level Optimization

We evaluated transaction-level optimization on the two convoluted coverage metrics of the DUT, which sit farther from test-constraints and failed to show accelerated coverage convergence using the above test pruning method in Section 6. A typical transaction in our testbench has 4 attributes "core, request type, address, and data" that can be randomized. The existing scoreboard logic is modified to capture the FSM and non-FSM based per transaction coverage.

The first transaction-level optimization to be tested is for FSM transition coverage metric: MESI design state-transitions. In this design, cache coherency is implemented using MESI protocol, i.e., a particular line in each L1 cache should be either in Modified, Exclusive, Shared or Invalid state. Overall the design MESI state is defined as combined MESI state for the corresponding cache line in all of the four cores, e.g., $\{I, M, I, I\}$ represents that the cache line in core 1 is in Modified state and Invalid for other cores. The coverage metric consists of a total of 143 valid state transitions. The design states are represented by 16 binary labels, 4 for each core. Hence, there are 19 input features (transaction data attribute not included) to the ML model and 16 binary output labels to classify. In phase 1, random tests are simulated until the training classification error is below 10%. In phase 2, coverage closure using online transaction pruning requires transaction attributes to be randomized in a cyclic random fashion to minimize redundant state transition predictions.

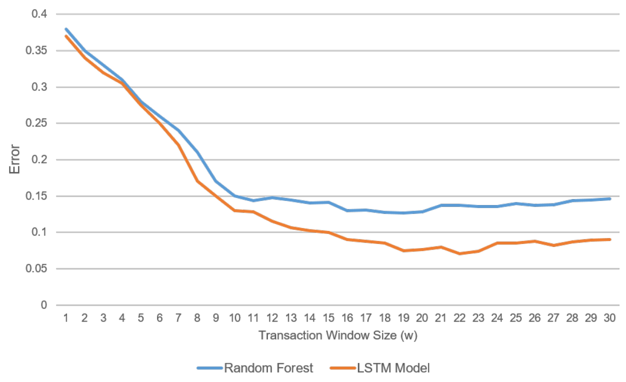


Fig. 5: Classification error with varying transaction window size (w) for LSTM model and RF classifier.

For transaction-level optimization on non-FSM event coverage, the event of interest is L1 cache hits on each address bin for each core. Such event occurrence depends on the last access of the same address and type of transactions simulated in the past. The cache hit coverage metric consists of a total of 768 bins. The RNN model is generated by forming a connected network of the LSTM (Long Short-Term Memory) units. The input attributes for feature extraction to the ML model are extracted from attributes of w consecutive transactions. For each transaction, 3 main attributes are considered and their values form a 22-bit wide input feature set: 4-core, 2-request type, and 16-address. Besides LSTM, we also implemented RF classifier for comparison. The training error threshold is again kept the same, $\gamma = 10\%$.

Figure 5 shows the training classification error as we vary the value of w. Smaller window size w results in inaccurate prediction due to insufficient attention to transaction history. On the other hand, too large w values cause over-fitting, and thus the training error increases slightly. For evaluation, the value of w picked is corresponding to the lowest point observed in the U-curve, i.e., $w = 20$. As the model with LSTM proved a higher prediction accuracy compared to the RF classifier, the latter is not assessed.

While performing online transaction pruning, newer transactions, with randomized attributes, are fetched from the stimulus generator component in the testbench. The transaction pruning decision is undertaken inside the driver. The driver code in the System-Verilog testbench is modified to export the transaction attribute values and parse them to a Python script for the coverage prediction. For offline sequence generation, the input attributes randomization is performed by the Python framework and only set of attribute values targeting coverage holes are translated into sequence of transactions for the simulation to achieve coverage closure.

The stimulus generation is done internally in online transaction pruning and only the decision about its effectiveness is made using prediction calls integrated into the driver. On the other hand, offline sequence generation includes some more sophisticated methods, like the generation of TA-graph, and requires more engineering efforts than just modifying the driver logic. This technique is also closer to coding a custom sequence for targeting coverage holes. The key difference is that instead of manual efforts, ML predictions are employed for such directed construction.

V. Experiment Results

A. Results for Test-Level Optimization

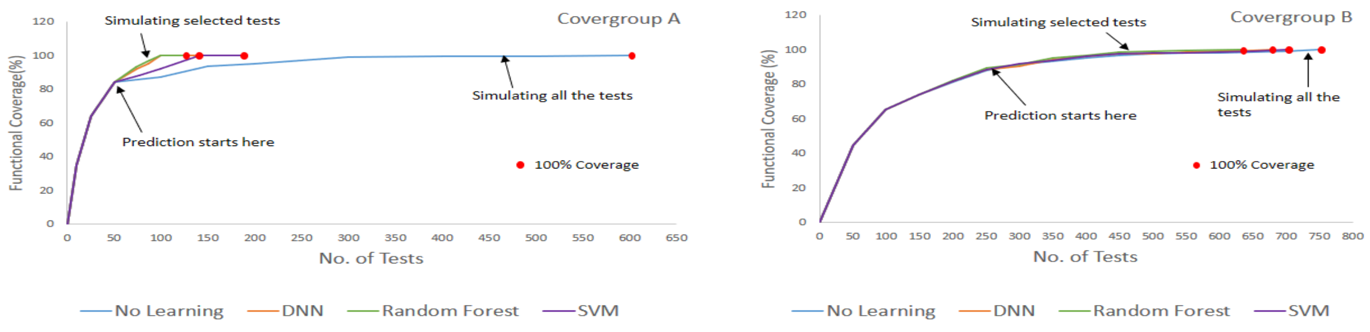


Fig. 6: Functional coverage closure for two groups with test-level optimization. Plot on the left shows coverage closure compressed towards left for cover-group A, while plot on right shows no significant left compression for cover-group B.

The eight metrics were divided into two cover-groups, based on their correlation with test-constraints. Cover-group A consists of six coverage metrics (827 bins) which are easy to reach by applying the right test constraints. Cover-group B contains the remaining two metrics (911 bins) which are fairly complex and do not have an obvious correlation with any test constraint. The results for both the cover-groups are plotted in Figure 6.

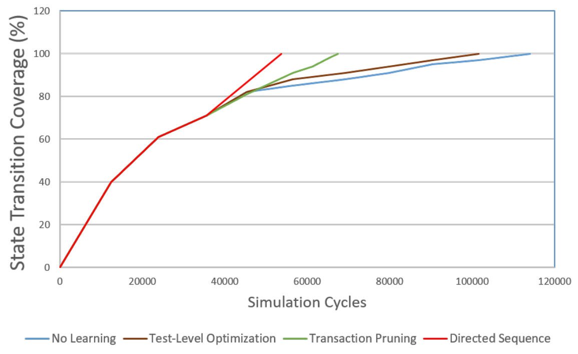
The training classification error for cover-group A was observed below 10% with simulation data from the initial 50 random tests, which proves these metrics have high linkage with test-constraints. Random tests regression took a total of 598 tests to obtain 100% coverage for cover-group A, although about 80% and 90% coverage marks were reached with 50 and 157 tests respectively, indicating a clear knee point on the coverage curve. The numbers of tests for coverage closure for different methods are summarized in Table I. The results from the conventional binary classification are also included. One can see that ML can considerably reduce the number of simulations. The advantage of using ternary classification is quite evident. Ternary classification-based ML can reduce the number of tests by $68\% - 78\%$. Among the three ML engines being evaluated, random forest and DNN are superior to SVM.

	No. of Tests for Group A Coverage Closure	
	Ternary ($\alpha = 0.9, \beta = 0.1$)	Binary ($\alpha = \beta = 0.5$)
No Learning	587	587
DNN	137 (77% $\downarrow$)	341 (41.9% $\downarrow$)
Random Forest	129 (78% $\downarrow$)	323 (44.9% $\downarrow$)
SVM	185 (68.5% $\downarrow$)	407 (30.6% $\downarrow$)

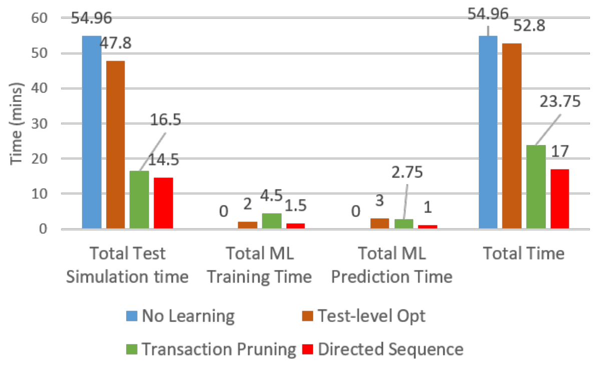
TABLE I: The numbers of tests simulated for verification closure for cover-group A.

The results on cover-group B, shown in Figure 6, are contrasting and reflect little benefit using test-level optimization technique. We observed that ML training for cover-group B is difficult to converge. Even after intensive training, due to high classification error on training data-set, the ML models still faced difficulty in deciding coverage. Thus, most of the bins fell into the undecided category and only a few tests were pruned. This reflects that cover-group B has a low correlation with test-constraints for coverage prediction. The result also reveals the limitation of test-level optimization and confirms the need for finer transaction-level optimization for the coverage metrics in cover-group B.

B. Results for Transaction-Level Optimization



(a) FSM transition coverage through random forest classifier.

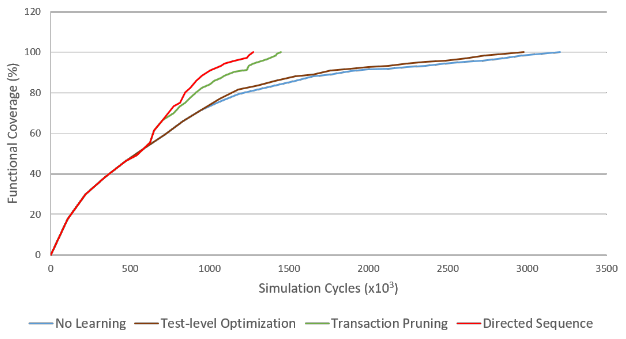


(b) CPU runtime for random forest classifier on FSM transition overage.

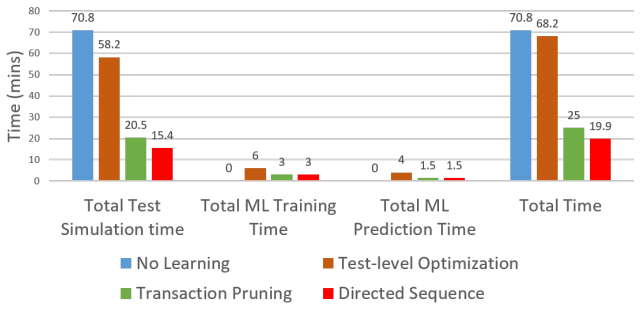
Fig. 7: FSM Coverage Metric

We applied FSM type transaction-level optimization for the design MESI state transitions- one of the coverage metric in cover-group B shown earlier. Figure 7a plot the simulation cycles required for coverage closure with fully random, test-level optimization, transaction pruning and directed sequence generation using RF classifier. With fully random regression, 80% coverage is reached in about 50K simulation cycles, while it takes an additional 65K cycles to reach 100% coverage. This implies that the coverage hits without ML have high redundancy. The reduction in simulation cycles using test-level optimization is small (about 13%) compared to transaction-level optimization. The transaction pruning technique and directed sequence generation technique can reduce simulation cycles by about 48% and 55%, respectively.

The reduction in simulation cycles comes with a cost of the time taken by ML model for training and prediction. The data pre-processing consumes negligible time and hence can be ignored. We observed that the training time of the random forest model is usually less than half of DNN. Therefore, we advocate a random forest-based approach. The total CPU time and its breakdown for the MESI state transitions metric are shown in Figure 7b. The overall training and prediction time are much



(a) Non-FSM event coverage through LSTM classifier.



(b) CPU runtime for LSTM-based technique on non-FSM event coverage.

Fig. 8: Non-FSM Coverage Metric

smaller than the simulation time. The results show reductions in total CPU time of about 57% and 69% with transaction pruning and directed sequence generation, respectively.

Figure 8a shows the results achieved by applying LSTM-based non-FSM type transaction-level optimization for the cache hit (event) coverage, also the part of cover-group B. The random regression and test-level optimization take approximately the same number of simulation cycles to reach 100% coverage. The LSTM model training error dropped below 10% when total coverage was 60%. The remaining 40% coverage was achieved with transaction pruning and directed sequence generation, which resulted in a reduction of 55% and 61% of simulation cycles, respectively, compared to random regression.

Figure 8b shows the total CPU runtime and its breakdown for evaluating the LSTM-based transaction-level optimization on non-FSM event coverage. Again, the LSTM training and prediction time are much less than simulation time. Compared to the random regression without machine learning and test-level optimization, coverage closure with transaction pruning and directed sequence generation can be reduced by 65% and 72% of the total CPU runtime, respectively.

Figures 7a, and 8a show a small improvement for offline sequence generation over transaction pruning in terms of simulation time-reduction when compared to total simulation time with no learning applied. Also, the total CPU time for both these methodologies are comparable. Hence, online pruning gives a remarkable run-time reduction with very few modifications to the framework while coverage closure can be compressed further left with a more intelligent methodology like offline sequence generation at the cost of additional engineering investment.

VI. Conclusion

Simulation-based verification is not the most optimized approach for functional verification for many reasons; lack of automated guidance from coverage data to mitigate the redundancy in randomized stimulus generation patterns being one of them. In this paper, we have explored and suggested a systematic approach to deploy machine learning for optimizing stimulus at different hierarchies to achieve accelerated coverage convergence. Simulation of directed tests is an alternate way for covering the leftover (complex and hard to hit) coverage in later stages of random regressions. Constructing such custom tests is a time intensive task and requires a thorough understanding of the design. Here, machine learning guided coarse- and fine-grained stimulus refinement techniques via test-level and transaction-level optimization, respectively are attractive alternatives for coverage closure.

A total of eight coverage metrics were considered for achieving speedup closure, each associated with different functional features of the design. ML classification capabilities were exploited for dual threshold categorization on each coverage bin as the basis for test pruning decision. Test-level optimization proved effective for six simple and moderately complex coverage metrics with a notable 78% reduction in the number of tests simulated for coverage closure. However, test-level optimization failed to provide significant gains for the other two metrics due to a lack of correlation with test-constraints.

Transaction-level optimization framework records and predicts per transaction activity, and thus it provides better control over generating stimulus patterns with a higher probability of hitting existing coverage holes. The implementation varies based on the type of target coverage metric (FSM or Non-FSM). Once the ML model is trained, the coverage predictions are utilized in two different ways: Online Transaction Pruning and Offline Sequence Generation, for the expeditious coverage closure. The latter is a more sophisticated method with marginally better results than the former technique. Overall, transaction-level optimization proves more effective and demonstrates around 70% reduction in the total CPU time required for verification coverage closure; while test-level optimization shows no significant gains for such metrics. Our work leads us to conclude that the complementary application of both of these techniques is the recommended path for efficiency improvements in functional verification coverage convergence.

References

[1] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using bayesian networks," *Design Automation Conference*, pp. 286–291, 2003.

[2] O. Guzeya, L.-C. Wang, J. R. Levitt, H. Foster, J. Bhadra, X. Feng, and M. S. Abadir, "Increasing the efficiency of simulation-based functional verification through unsupervised support vector analysis," *Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, pp. 138–148, 2010.

[3] W. Chen, N. Sumikawa, L.-C. Wang, J. Bhadra, X. Feng, and M. S. Abadir, "Novel test detection to improve simulation efficiency – a commercial experiment," *International Conference on Computer-Aided Design*, vol. 29, pp. 101–108, 2012.

[4] C. Ioannides and K. Eder, "Coverage-directed test generation automated by machine learning - a review," *ACM Transactions on Design Automation of Electronics and Systems*, vol. 17, pp. 7:1–7:21, 2012.

[5] S. Sokorac, "Optimizing random test constraints using machine learning algorithms," *Design Verification Conference*, 2017.

[6] F. Wang, H. Zhu, P. Popli, Y. Xiao, P. Bodgan, and S. Nazarian, "Accelerating coverage directed test generation for functional verification: a neural network-based framework," *Great Lake Symposium on VLSI*, pp. 207–212, 2018.

[7] R. Roy, C. Duvedi, S. Godil, and M. Williams, "Deep predictive coverage collection," *Design Verification Conference*, 2018.

[8] I. Wagner, V. Bertacco, and T. Austin, "Stresstest: an automatic approach to test generation via activity monitors," *Design Automation Conference*, pp. 783–788, 2005.

[9] O. Guzey, L.-C. Wang, J. Levitt, and H. Foster, "Functional test selection based on unsupervised support vector analysis," *Design Automation Conference*, pp. 262–267, 2008.

[10] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, pp. 1735–1780, 1997.

[11] Chollet, François, *et al.*, "Keras," *https://keras.io*, 2015.

[12] Fabian, Gaël, *et al.*, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[13] A. Liaw and M. Wiener, "Classification and regression by random forest," *R News*, vol. 2/3, pp. 18–22, December 2002.