Deep Predictive Coverage Collection

Rajarshi Roy rajarshir@nvidia.com
Chinmay Duvedi cduvedi@nvidia.com
Saad Godil sgodil@nvidia.com
Mark Williams mwilliams@nvidia.com
NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050

***Abstract*- Coverage metrics are critical in evaluating and guiding the functional verification process of complex hardware designs. However, collecting certain coverage metrics can add a significant runtime overhead to the register-transfer-level (RTL) simulation of a design. Since running large batches of tests with RTL simulation is the primary process of hardware verification, it is of interest to reduce the runtime overhead introduced by collecting coverage metrics. In this paper, we propose a method to achieve this goal by employing two machine learning techniques: k-means clustering and deep neural networks (DNNs). K-means clustering determines a small subset of the design to collect coverage for and DNNs predict the coverage of the rest of the design from the collected coverage. Experimental results on the per-module condition coverage metric of various NVIDIA hardware design units demonstrate that the metric can be reported with low error rates of 5% while collecting coverage for only 3% of the modules in the design. For one unit, the simulation runtime overhead of collecting coverage reduced from 100% to 10% while reporting the coverage metric within 3% error.**

I. Introduction

The process of hardware design verification is often guided by coverage metrics [1]. During the functional verification cycle, coverage metrics provide essential feedback on the extensiveness of tests in covering various hardware functionality during simulation. The lack of coverage in any area of the hardware functionality guides the verification process to stress those areas with updates to the testbench. Certain common coverage metrics such as line and condition coverage provide visibility into how extensively each portion of the hardware design is tested. Line coverage provides a feedback on whether tests are reaching every line in the register-transfer-level (RTL) design of the hardware during RTL simulations. Condition coverage provides a feedback on whether tests are exercising every conditional decision in the RTL. Other types of coverage metrics include functional coverage where specific functional scenarios are measured for reachability. Furthermore, functional scenarios can be automatically generated for coverage measurement from the state-space of the design such as toggles and finite-state-machines (FSM).

The RTL simulation runtime for large and complex hardware designs can be long. Simulation for certain tests can run for multiple hours to days. In our experiments, we observed that collecting coverage metrics can add a significant overhead to the simulation runtime (Figure 5). In practice, coverage is collected for tests only during dedicated coverage runs because of the runtime overhead. As a result, there is no feedback on testbench improvements or regressions between the dedicated coverage runs. Thus, a low-overhead coverage feedback between full coverage runs could greatly benefit the verification process.

Previous work [2] analyzing coverage data across modules in a hardware design for a batch of tests had observed high correlation across the modules and potentially high correlation across tests. The cross-test coverage correlation has also been utilized to create small suites of tests from a large set of tests with high redundancy using clustering [3]. We focused on the complementary problem of utilizing the cross-module coverage correlation to select a small subset of modules with essential coverage data. We demonstrate that constraining our coverage flow to collect coverage only for the subset of modules can significantly reduce the RTL simulation runtime with coverage collection (Figure 5), making it practical for this degree of coverage collection to be enabled for all test runs. Furthermore, we focused on utilizing the cross-module coverage correlations to predict the coverage for the rest of the design outside the subset with low error. This allows for a low-error proxy for the full coverage report for all tests.

In the overall flow, an updated subset of modules is selected with the data from every full coverage run. The cross-module correlations in coverage are also learnt on this data. In all the tests that are run before the next full coverage run, coverage is collected only on the subset of modules and the learnt correlations infer the coverage of all modules. The flow can be automatically validated for the integrity of predictions by inferring the coverage of the next full coverage run from the currently selected subset of modules and comparing with the actual coverage values. Between full coverage runs, if a module outside the coverage collection subset is reported to have low coverage, the module can be added to the subset for coverage collection and debug. This additional data will not disrupt the inference process.

II. Background

A. K-Means Clustering

K-means clustering [4] is a popular algorithm for cluster analysis and data mining. It is an efficient unsupervised algorithm to detect patterns in a dataset and to select a small subset of data points that represent these general patterns. In k-means clustering, we represent our data as a set of n m-dimensional vectors $\{x_1, ...x_n\}$ that we want to group into k clusters. The algorithm proceeds as follows:

1. Randomly initialize k cluster centroids $\{c_1, ... c_k\}$
2. Repeat until convergence:
    a. For each x_i, assign a label $l_i = j$ such that c_j is the closest centroid to x_i
    b. Update the centroids such that c_j is now the centroid of all x_i's such that $l_i = j$

At the end of each iteration, the total error is defined as the sum of the Euclidean distance between each vector and its corresponding centroid. The algorithm is considered converged when there are no updates in centroid assignments in the last iteration.

B. Deep Neural Networks

Since efficient training methods [5] led to neural networks models successfully recognizing handwritten digits with high accuracy in 1990 [6], advances in parallel computation and neural network architectures have led to much larger and sophisticated "deep" neural network models that can solve complex image recognition [7] and speech recognition [8] tasks. A single deep neural network (DNN) model can learn complex relations from a set of inputs to a set of outputs given example input set-output set pairs. After learning, the trained deep neural network can efficiently predict the output set when provided with a new input set. The ability to learn and predict multiple output values with a single model makes the DNN attractive over other machine learning models. Furthermore, a DNN can recognize multiple outputs that are correlated and accordingly share the same resources (neuron weights) for the correlated outputs. This allows the DNN model to scale efficiently to a large number of outputs. This remainder of this section will provide a brief overview of neural network architecture and training algorithms.

The fundamental building block of a neural network is a simplified mathematical model of a biological neuron. A neuron takes a vector of real numbers as input and outputs a single real number that is a bounded, non-linear, parametrized function of this vector. Hence, each neuron computes:

$$Y = f(W^T X + b). \quad (1)$$

In this equation, X is the input vector to the neuron, W is a matrix of weights, b is a vector of biases, f is a non-linear function like sigmoid, tanh or ReLU and Y is the output of the neuron. W and b represent the parameters of the neuron that are learnt during the training phase. The function f is usually considered as a separate neuron called the activation neuron.

These neurons stacked together in multiple layers such that the output of layer n is the input to layer $n+1$ to form a neural network. The goal of training a neural network is to derive the best set of parameters for all the neurons in a neural network that maximizes the likelihood of the output of the neural network being closest to the expected output. Different DNN architectures are characterized by the way consecutive layers of the DNN are connected to each other. An architecture where the outputs of all the neurons of layer n are provided as inputs to all the neurons of layer $n+1$ is said to be comprised of "fully connected" layers or "dense" layers. Layers comprised of activation neurons are called activation layers.

Training of a neural network is carried out in two steps. The first step is called the forward propagation step. In this step, input data is fed to the first layer and flows through the connections between layers till the last layer. Each layer takes the output of the previous layer as its input, computes its own output according to (1) and feeds it forward to the subsequent layer. The output of the last layer is the output predicted by the neural network. This is followed by the error back-propagation step. In this step, the output of the last layer is compared against the expected output and a loss function that represents the error in prediction is calculated. The goal of the training process is to minimize this loss. This process exploits the fact that the function computed by each neuron is differentiable. So, the gradient of the overall loss function can be computed with respect to each parameter of each neuron in the neural network by using the chain rule of differentiation [5].

Once these gradients have been computed, each parameter is updated in the direction of descending gradient. The size of the step taken in the direction of descending gradient is called the "learning rate". Typically, we begin the training process with a large learning rate and gradually decay the rate for subsequent training iterations [5]. The learning rate and other factors that affect the training of a neural network are referred to as hyperparameters. Typically, training data is broken down into multiple batches. All the training data in one batch goes through the feed forward step, with the loss being accumulated over the entire batch. Then, error back-propagation is carried out on this accumulated loss [5].

Finally, a recent technique called batch-normalization [9] greatly reduces the sensitivity to initial parameter weights and learning rate. This allows network depth to scale without the need for careful tuning. The batch-normalization layer of neurons learns to normalize the outputs of dense layer towards a mean of 0 and a standard deviation of 1 which ultimately prevents the problem of exploding gradients during backpropagation that saturates activation layer neurons [10].

III. METHODOLOGY

Our goal is to reduce the runtime overhead of coverage collection during RTL simulation while still obtaining accurate coverage data. There are two steps to our overall method:

1. We use k-means clustering to determine a subset of modules that best represents the design. The modules closest to the cluster centroids are referred to as 'sampled modules' in the rest of the paper. The rest of the modules are referred to as 'unsampled modules'.
2. Then a DNN model can be trained to predict the coverage for unsampled modules using just the coverage for sampled modules. Once the DNN has been trained, we only need to measure the coverage for the small subset of sampled modules while the coverage for unsampled modules can be predicted from the sample module coverage.

As discussed in the introduction section, these steps can be integrated into the overall verification flow with full coverage runs at regular intervals providing the data for k-means clustering, training the DNN model and validating the accuracy of the method itself.

A. Dataset

The data used in our experiments was collected from full coverage runs of multiple RTL units. These units were of varying sizes (Table 1) and had varying complexity in their designs. For this study, we conducted our experiments on overall module level coverage metrics instead of reachability at the specific line, condition or functional coverpoint granularity. Thus, for a unit with m modules, there were *m* datapoints collected per test, one for each module, in the suite of *t* functional tests in the coverage run. Each datapoint would reflect the fraction of reachability conditions in a module that were reached during the RTL simulation of a test. For example, if a test covered two out of five lines in a module, the datapoint gathered would be 40% line coverage for that particular test and module.

TABLE 1
PER UNIT MODULE (INSTANCE) COUNTS

	Counts
Unit A	82
Unit B	121
Unit C	176
Unit D	224
Unit E	268
Unit F	410
Unit G	500
Unit H	574

B. Metrics

We measure the effectiveness of our approach using the following metrics:

1. Prediction error for unsampled modules.

The effectiveness of our approach is primarily evaluated by how accurately it can predict coverage for unsampled modules. Hence, we measure the Mean Absolute Error (MAE) in coverage prediction for all modules in the design. We calculate the mean absolute error across all modules across all tests. The MAE metric is formally defined in (2).

$$MAE = \frac{\sum_{i=1}^{m}\sum_{j=1}^{t} abs(predcov_{ij} - cov_{ij})}{m * t} \quad (2)$$

m = number of modules in the design
t = number of tests in the test suite
$predcov_{ij}$ = the predicted coverage for the module i with test j
cov_{ij} = the actual coverage for the module i with test j

The MAE across the all modules and across all tests can be directly interpreted as the average value by which predicted coverage values deviated from the actual coverage values. We do not choose the root-mean-square (RMS) error metric because of its sensitivity to outliers [11].

We want to ensure that the error metric does not get biased by the number of lines (or conditions) in a module. So, we do not scale the error in coverage prediction for a module by the number of lines (or conditions) in that module.

Finally, another detail in calculating the MAE error is that the mean error is calculated across all modules and not just the unsampled subset. This is to maintain consistency of the metric across the range of sampled subset size. As a result, if the set of sampled modules is 100% of all modules, the MAE is 0% instead of being undefined. This reflects the usecase because all coverage being collected for all modules does mean that there will be no error in representing the coverage of the modules.

2. Sampled Module Count %

The number of modules that get sampled alone is not a good indicator of the effectiveness of our approach because the total number of modules can vary across design units. Instead, we measure the number of sampled modules as a percentage of the total number of modules in the design. This metric is formally defined in (3).

$$Sampled\ Module\ Count\ \% = \frac{\#Sampled\ modules}{\#Total\ modules} * 100\ (3)$$

3. Simulation runtime overhead

We recognize that some designs are organized with a lot of code in a few modules. So, if these modules with most of the code are selected as sampled modules, Sampled Module Count % will be low, but the simulation run time will not be affected proportionately. To quantify this effect, we measure the increase in simulation runtime when coverage is collected for all modules as well as for just sampled modules and compare these overheads.

C. Sampled Module Subset Selection

To find an effective subset of modules which best represents the coverage of all modules, there could be various methods. We found that very few modules had exactly the same coverage across all tests. Thus, the subset of modules in a unit with unique coverage values across all tests were almost as large as all the modules in the unit. Likewise, after rank analysis of the data, we observed that the subset of linearly independent modules was also almost as large as all the modules in the unit. Thus, we selected a clustering approach whereby modules with very similar coverage behavior are clustered together. To find the "essential" subset of modules, we select one module from each cluster since two modules from the same cluster contribute roughly the same coverage information. For the clustering step, we used the k-means clustering algorithm in the scikit-learn python library [12]. In the k-means clustering framework, each datapoint corresponds to a module and the number of dimensions per datapoint is equal to number of tests in the dataset.

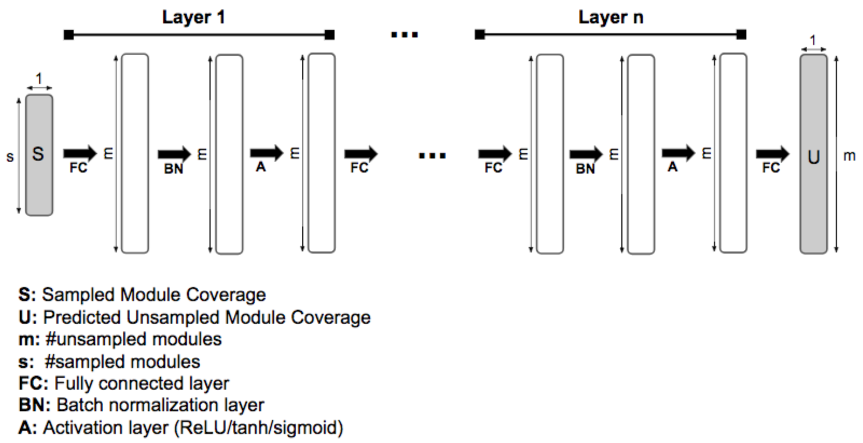
Since the mean value of each cluster minimizes the (Euclidean) distance to all datapoints in a cluster [4], we select the module that has the coverage behavior that is closest (Euclidean distance) to the mean coverage behavior per cluster. For each cluster, this selected module will be referred to as the cluster centroid. The k cluster centroids from the k clusters of modules after running the k-means clustering algorithm will be the sampled module set for which coverage will be collected. Coverage will be predicted for the remaining modules using methods described in the next section.

D. Unsampled Module Coverage Prediction

After we collect coverage for the sampled modules, we must estimate the coverage for the unsampled modules. We experiment with two prediction methods to estimate the unsampled module coverage: the centroid assignment (CA) prediction and the deep neural network (DNN) based prediction.

With the centroid assignment method, the coverage for an unsampled module in a test is simply predicted to be the coverage of the cluster centroid of the cluster it belongs to. However, the coverage for the unsampled module could possibly be more effectively determined from the coverage of all the sampled modules. The DNN method uses a deep neural network that takes in the coverage of all sampled modules as input and learn to predict the coverage of all unsampled modules.

Our DNN architecture contains fully connected input and output layers. The size of the input layer matches the number of sampled modules whose coverage is available while the size of the output layer matches the number of unsampled modules whose coverage is to be predicted. To model complex relations between the sampled and the unsampled module coverage, there are several "hidden" layers which consist of fully connected, batch-normalizations and activation neurons (Figure 1). The number of hidden layers were tuned for optimality between a range of two to eleven layers. The activation neuron was chosen for optimality among sigmoid, tanh and ReLU functions. The training procedure minimized the L2 loss function with the Adam optimizer algorithm using the Tensorflow library [13].



Layer 1
Layer n
S
U
s
m
1
E
FC
BN
A
S: Sampled Module Coverage
U: Predicted Unsampled Module Coverage
m: #unsampled modules
s: #sampled modules
FC: Fully connected layer
BN: Batch normalization layer
A: Activation layer (ReLU/tanh/sigmoid)

Figure 1. Deep neural network architecture for predicting the coverage of all unsampled modules given the coverage of all sampled modules.

Two naïve baselines that utilize modules with low test-to-test coverage variance were also evaluated. These baselines are the average assignment (AA) and selective assignment (SA) baselines. The average assignment baseline records the average coverage per module from the training tests and simply predicts the coverage to be the recorded average value. This could be more effective than the cluster centroid assignment for modules that have very low test-to-test variance. The selective assignment baseline observes from the training tests whether the average assignment or cluster assignment was a better estimator per module and makes the same choice for validation test coverage prediction.

IV. Experimental Results

In practice, the collection of both the line and condition coverage metrics add significant simulation runtime overhead because they track a large number of reachability events. Both metrics are likely to have similar cross-module coverage correlation since their reachability events are related to the RTL implementation of the design. We chose the condition coverage metric for our experiments because of the availability of the data from the various testbenches.

Per-module condition coverage was collected for all modules in the eight units described earlier (Table 1) using a diverse suite of functional tests. The collected coverage data was split into two random nonoverlapping subsets. The training set contained coverage from 40% of the tests and the validation set contained coverage from 60% of the tests. The number of tests run for each unit are listed in Table 2.

TABLE 2
Training And Validation Test Counts

	Train (40%)	**Validation (60%)**	**Total**
Unit A	13416	20124	33540
Unit B	5864	8796	14660
Unit C	8612	12918	21530
Unit D	20676	31014	51690
Unit E	3964	5946	9910
Unit F	3772	5658	9430
Unit G	12752	19128	31880
Unit H	13224	19836	33060

For every unit, the training test set was used to select the sampled module subset and to learn the coverage behavior of all the modules. Then coverage was sampled from the validation test set for only the sampled module subset. Finally, the coverage for the unsampled modules in each test from the validation set was predicted using the coverage for the sampled modules in that test and the learned coverage behavior of all modules. The predicted coverage was then compared to the actual coverage in the validation set for error evaluation.

A. Clustering Analysis

To understand the behavior of k-means clustering for our dataset, we ran the k-means clustering algorithm on the training dataset for a range of increasing k values starting from k=1. For each k value, we applied the centroid assignment (CA) method to predict the coverage of the unsampled modules from the k sampled modules in the validation set. To observe the clustering behavior at a detailed granularity, we visualized the mean absolute error (MAE) per module. The modules were ordered by increasing prediction MAE within each cluster.

From the visualization presented in Figure 2 we made several observations on how k-means clustering clusters modules for increasing values of k. At k=1, a cluster centroid is found such that all modules can be predicted to have the cluster centroid's coverage with less than 60% MAE. With increasing k values from 1 to 4, the single cluster is broken up into smaller clusters that are approximately uniformly sized (number of modules). At k=10, a single cluster has formed with modules that have the same coverage behavior. Thus, the centroid assignment error for that cluster is close to zero. From k=10 to k=20, the clusters with close to zero error remain largely intact and outlier modules from another clusters breakaway to form their own cluster. At k=20, there is a wide range of cluster sizes, each consisting of modules of similar coverage behavior and thus having low centroid assignment MAE.

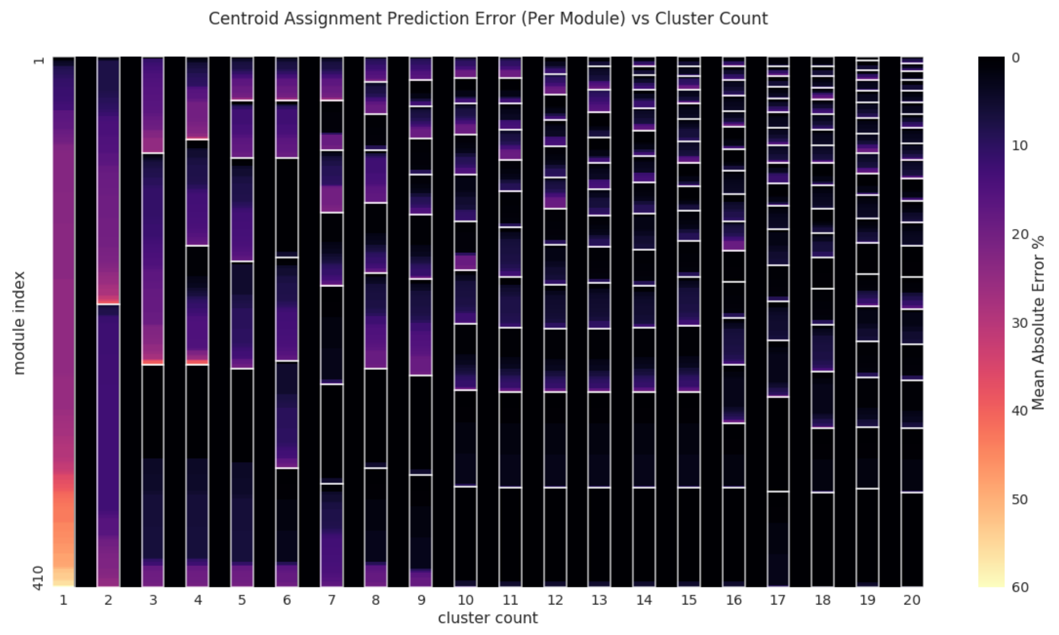


Figure 2. Per module MAE for the centroid assignment prediction method using centroids from k-means clustering with k values ranging from 1 to 20. Each column corresponds to a k value and each row corresponds to a module. The module order is sorted by MAE within each cluster. The span of each cluster is highlighted by white rectangles. MAE is visualized as heatmap. The data corresponds to Unit F's condition coverage.

B. Prediction Error

With the training set of tests for every unit, we performed a k-means clustering sweep of k values ranging from 1 to the number of modules in each unit. For each k value, the sampled module set corresponding to the k cluster centroids were saved. Keeping the sampled module set fixed, all the techniques to predict unsampled module coverage were evaluated on the validation set. The two prediction methods evaluated were the deep neural network (DNN) based prediction and the centroid assignment (CA) prediction. The average assignment (AA) and selective assignment (SA) naïve baselines were also evaluated.

Since deep neural nets must be tuned for specific input and output sizes for the highest accuracy, we focused the DNN evaluation to sampled module set sizes (k values) ranging from 1 module to 10% of the total module count per unit. Beyond finding the sampled module set and cluster assignments, the training set was also used to train the neural networks and tune their hyperparameters.

The mean absolute error (MAE) evaluations for all the prediction methods across all the units (Figure 3) yielded several consistent trends. Firstly, at low sampled module proportions (0.5% to 2%), the centroid assignment MAE was higher than the average assignment MAE. This indicates that in these ranges of sampled module proportions, there are fewer unsampled modules that map their coverage to the sampled modules than to their own average coverage value. However, in this range the selective assignment method has lower MAE than the average assignment method, which indicates that the sampled modules still have some useful information. At higher proportions of sampled modules (10%), the centroid assignment MAE steadily decrease to a range of 7.3% (Unit A) to 1.5% (Unit F) (Table 3). This result demonstrates that k-means clustering followed by centroid assignment is a powerful technique in coverage prediction on its own right. This result is also consistent with the per module MAE analysis observations made earlier.

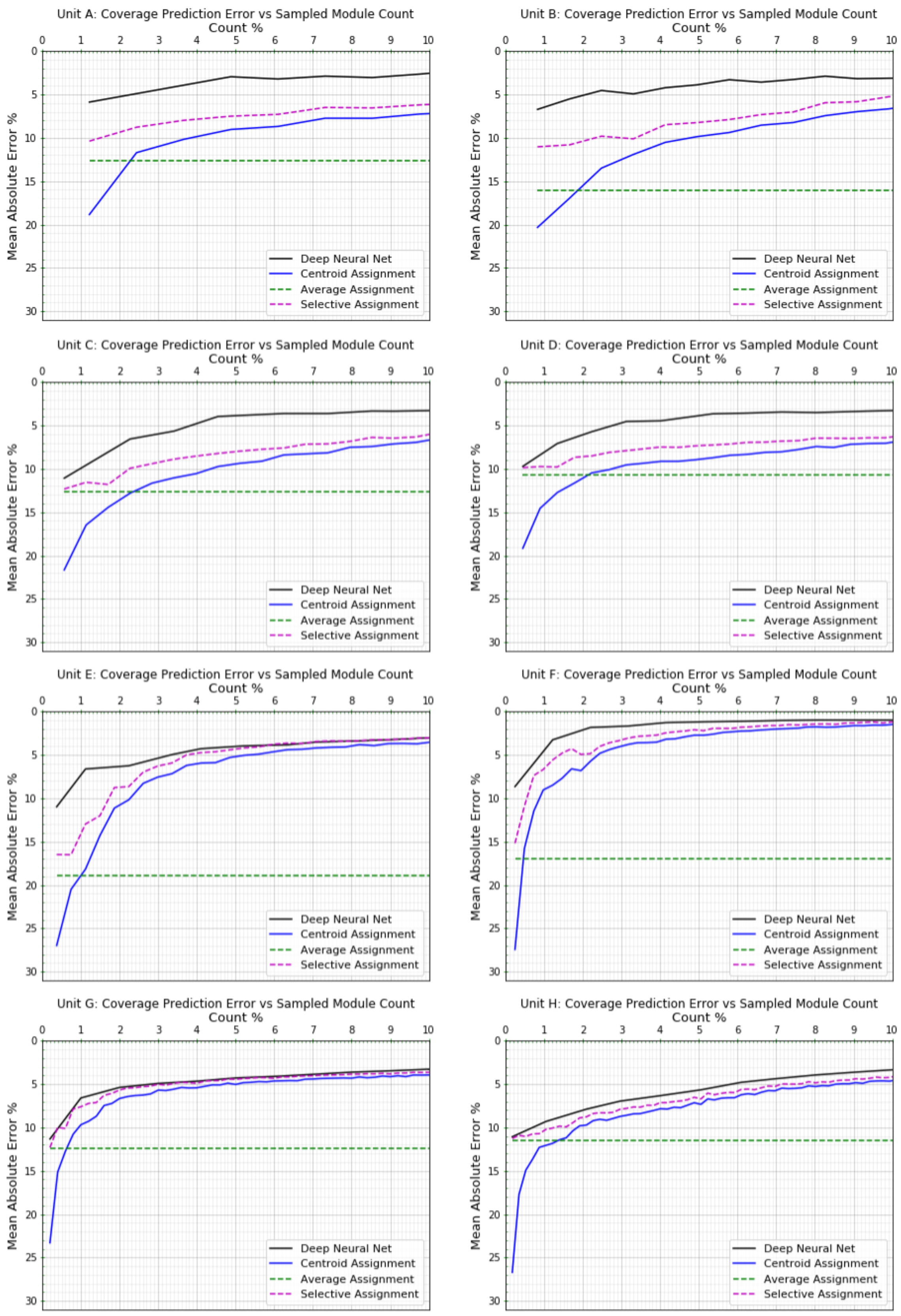


Figure 3. Per-unit coverage prediction error on validation tests.

The deep neural net (DNN) based prediction performed consistently better than the centroid assignment prediction with lower MAE throughout the sampled module range. The MAE of the DNN methods decreased to a range of 3.9% (Unit H) to 1.0% (Unit F) at 10% of the modules sampled (Table 3). The DNN prediction method also performed consistently better than the selective assignment baseline, indicating that the DNN made better inferences about the coverage of the unsampled modules than simply deciding on predicting the cluster centroid versus the average coverage value.

TABLE 3
PREDICTION ERROR (MAE) WITH 10% OF MODULES SAMPLED

	DNN	**CA**
Unit A	2.8%	7.3%
Unit B	3.1%	6.7%
Unit C	3.4%	7.2%
Unit D	3.4%	7.2%
Unit E	3.4%	3.9%
Unit F	1.0%	1.5%
Unit G	3.3%	3.9%
Unit H	3.9%	4.7%

Since the DNN prediction method achieved a less than 5% error across all units, we compare the minimum sampled module proportion to achieve 5% error (MAE). We can observe from this comparison (Table 4) that the DNN method only needs 3.0% of the modules to be sampled to predict the coverage for all modules with less than 5% error on average. The centroid assignment method needs 12.0% of the modules sampled in comparison. For errors below the 10% bound, the DNN method needs 0.7% of modules to be sampled while the centroid assignment method needs 2.7% of modules to be sampled.

TABLE 4
NECESSARY PERCENTAGE OF MODULES SAMPLED TO ACHIEVE 5% AND 10% ERROR BOUNDS

	Sampled module % for 10% MAE		**Reduction factor**	**Sampled module % for 5% MAE**		**Reduction factor**
	DNN	CA		DNN	CA	
Unit A	1.2%	3.8%	3.2	2.3%	21.3%	9.2
Unit B	0.8%	4.8%	5.8	2.1%	15.0%	7.2
Unit C	1.0%	4.4%	4.5	3.8%	17.7%	4.6
Unit D	0.4%	2.8%	6.2	2.8%	20.7%	7.5
Unit E	0.5%	2.3%	4.2	3.3%	5.4%	1.6
Unit F	0.2%	0.9%	3.6	0.9%	2.4%	2.6
Unit G	0.4%	0.9%	2.2	2.8%	4.7%	1.7
Unit H	0.7%	1.9%	2.6	5.9%	8.6%	1.5
Mean	**0.7%**	**2.7%**	**4.0**	**3.0%**	**12.0%**	**4.5**

Finally, analyzing the reduction in the proportion of sampled modules when using the deep neural network based method as opposed to the centroid assignment method (Figure 4) reveals that for the 10% error bound, there is a 2.2x (Unit G) to 6.2x (Unit D) reduction. For the 5% error bound, there is a 1.5x to 9.2x reduction.

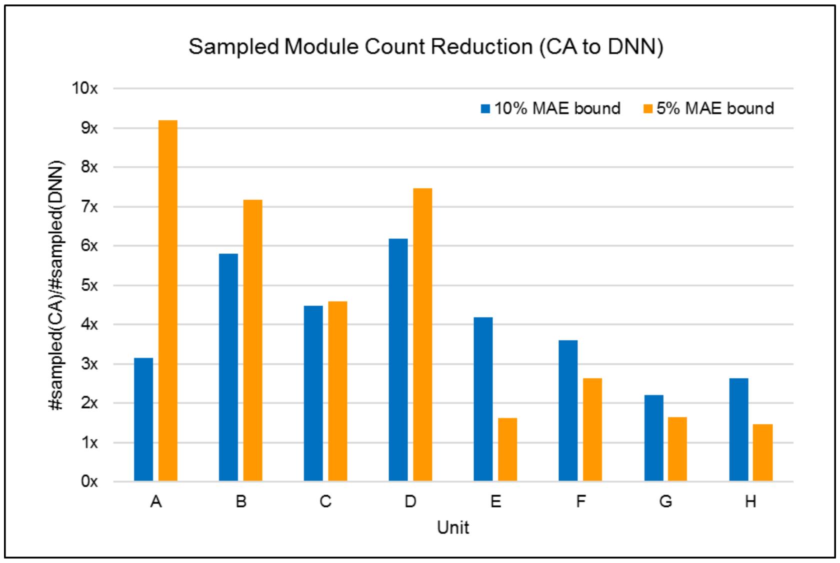


Figure 4. Reduction in the sampled module count at fixed error bounds when using deep neural net based prediction versus centroid assignment based prediction.

C. *Runtime Overhead Savings*

Simulating RTL while collecting coverage results in a significant increase in simulation run time. To evaluate the speedup of collecting coverage for the sampled module subset instead of all modules (full coverage), we benchmarked three metrics for each unit using a variety of tests.

- Runtime of simulation with no coverage collected
- Runtime of simulation with full coverage collected (all modules)
- Runtime of simulation with coverage collected for only a subset of modules

The sampled module subset chosen for this experiment was same set of the 10% modules sampled to evaluate the error in (Table 3). These subsets could be used to predict the coverage for all modules with a minimal 1.0% to 3.9% mean absolute error.

After normalizing the simulation per unit so that the simulation runtime with full coverage is 100%, we observe (Figure 5) that the runtime with the 10% coverage subset collected consumes 50% (Unit H) to 87% (Unit B) of the full coverage runtime. The simulation runtime without coverage collection for the corresponding units were 28% (Unit H) to 80% (Unit B). This corresponds to the simulation overhead of collecting coverage from the 10% subset being 30.5% (Unit H) and 35.0% (Unit B) of the simulation overhead of full coverage. The best reduction in simulation overhead was for Unit A where the overhead for collecting coverage from the 10% subset was also 10% of the overhead of collecting full coverage. The prediction accuracy at this sampled module proportion was 2.8%.

Further benchmarking and analysis would be required to determine why the overhead was not around 10% across all units. A possible factor could be that 10% of modules sampled were more complex than the remaining 90% of unsampled modules. In that case, the clustering and centroid picking algorithm could be modified to weigh down complex modules within additional error margins. Another factor could be that there is an overhead to enable coverage collection that is independent of the portion of the design that coverage is being collected for.

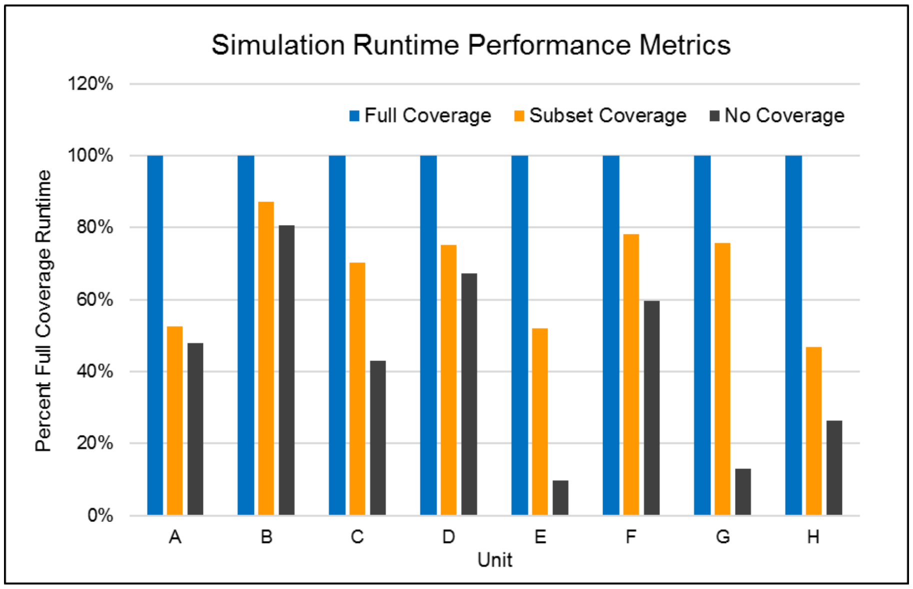


Figure 5. Per unit simulation runtime benchmarks for collecting coverage for all modules, a 10% subset of modules and no modules.

IV. Conclusion And Future Work

In this project we explored a new approach to collect coverage for an essential subset of modules in each unit and to predict the coverage for the rest of the modules in the unit using deep neural networks. This approach allowed us to collect coverage for only 3% of the modules to make a prediction with less than 5% error averaged across a diverse set of units. For a unit, we conducted preliminary simulation runtime benchmarking and achieved a 10x reduction in simulation runtime overhead while estimating coverage with 2.8% mean absolute error.

The existence of cross-module coverage correlations for the per-module condition coverage metric allowed the approach to be effective. We believe that the approach could be similarly effective for the per-module line coverage metric due to similar correlations. It is not obvious whether other types of functional coverage such as toggle, finite-state-machine or coverpoint coverage will have similar correlations. Large cross-products in cross-coverage coverpoints might possibly benefit from correlations but further experimentation is required to confirm this claim.

Furthermore, it may not be necessary to apply this approach with these types of coverage metrics if they do not add significant simulation runtime overhead.

We wish to further benchmark simulation runtimes to understand the factors involved in coverage collection runtime overheads. We also wish to benchmark simulation runtimes while collecting coverage for smaller subsets of modules that can achieve 5% error margins. Finally, we hope to scale up coverage estimation at a per-line or per-condition granularity. Since the coverage at that granularity is essentially binary and could have redundancies in the form of Boolean dependence, a promising research direction would be to use a combination of logic minimization methods [14] and lightweight binarized neural networks [15].

ACKNOWLEDGMENT

We would like to thank our employer, NVIDIA Corporation for supporting this research. We would also like to thank the organizers of the Design and Verification Conference (DVCON) for their constructive feedback and the opportunity to share our work.

REFERENCES

[1] A. Piziali, "Coverage Driven Verification" in *Functional Verification Coverage Measurement and Analysis*, Kluwer, 2004, pp. 109-136.

[2] M. Farkash, B. Hickerson, M. Behm and B. Samynathan, “Mining Coverage Data for Test Set Coverage Efficiency” in *Design and Verification Conference, DVCON 2015, Santa Clara, CA, USA*, 2015.

[3] S. Ikram and J. Ellis, “Dynamic Regression Suite Generation Using Coverage-Based Clustering” in *Design and Verification Conference, DVCON 2017, Santa Clara, CA, USA*, 2017.

[4] J. A. Hartigan and M. A. Wong, "Algorithm AS 136: A k-means clustering algorithm" in *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28, no. 1, 1979, pp. 100-108.

[5] Y. LeCun, L. Bottou, G. B. Orr and K. R. Müller, "Efficient backprop" in *Neural networks: Tricks of the trade*, Springer, Berlin, Heidelberg, 1998, pp. 9-50.

[6] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard and L. D. Jackel, "Handwritten digit recognition with a back-propagation network" in *Advances in neural information processing systems*, 1990, pp. 396-404.

[7] A. Krizhevsky, I. Sutskever and G. E. Hinton, "Imagenet classification with deep convolutional neural networks" in *Advances in neural information processing systems*, 2012, pp. 1097-1105.

[8] D. Amodei, A. Sundaram, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper et al, "Deep speech 2: End-to-end speech recognition in english and mandarin" in *International Conference on Machine Learning*, 2016, pp. 173-182.

[9] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift" in *International Conference on Machine Learning*, 2015, pp. 448-456.

[10] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks" in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 2010, pp. 249-256.

[11] P. J. Huber, E.M. Ronchetti, "Robust statistics" in *International Encyclopedia of Statistical Science*, Springer Berlin Heidelberg, 2011, pp. 1248-1251.

[12] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel et al, "Scikit-learn: Machine learning in Python" *Journal of Machine Learning Research*, 2011, pp. 2825-2830.

[13] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado et al, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems" *arXiv preprint arXiv:1603.04467*, 2016.

[14] R. K. Brayton, G. D. Hachtel, C. McMullen and A. Sangiovanni-Vincentelli, *Logic minimization algorithms for VLSI synthesis*. Vol. 2. Springer Science & Business Media, 1984.

[15] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1" *arXiv preprint arXiv:1602.02830*, 2016.