mL: Shrinking the Verification volume using Machine Learning

Yash Phogat, Patrick Hamilton
Arm Inc., Austin TX, USA
yash.phogat@arm.com, Patrick.Hamilton@arm.com

***Abstract*- Verifying a complex hardware design is a compute-intensive task. Design verification often uses a constrained random [1] approach to simulate tests using random stimuli. However, due to the random nature of this approach, efficiency diminishes over time, i.e., mature designs waste compute cycles in running tests that pass. Using Machine Learning (ML) we model the tests' history to build classification models that filter the tests with high likelihood of passing. To do this, the machine learns to identify(model) the parts of the test input stimulus which show high correlation to the fail/pass label. By applying the learnt model to a new set of tests, we obtain a filtered subset of tests that are prone to failure. Running this smaller set of tests on simulators will cut down our regression size and increase regression efficiency. In this paper, we discuss the ML modelling technique, its application to a hardware design project, compare ML performance against existing baseline (constrained random).**

Keywords - Machine Learning, Constrained Random Testing (CRT), Hardware Verification, Fail Signature, Supervised Learning, Cross Validation, Preprocessing, Inference, Regression

I. Introduction

A. *Hardware Verification using Constrained Random Testing*

A typical modern CPU verification testbench has many test controls such as configurations, scenarios, stimulus randomizations, checker modes etc. commonly referenced to as *knobs. For the scope of this paper, we will be discussing only pre-simulation randomization. Simulation time randomizations can also be influenced by ML but that is a different more complicated ML application where prediction happens during simulation.* For a typical hardware design, pre-simulation randomizations result in large number of knobs (order of thousands). That would result in an exponential number of all possible knob **combinations**, which would require impractical amounts of compute to run (simulate) exhaustively. Hence, a commonly used verification approach is Constrained Random Testing (CRT) [1], where the knob values are randomly picked, and tests are generated. One may ask, how is this approach "Constrained Random" and not just "Random"?

Verification engineers have design specs and past verification experience based on which they introduce constraints within their testbench. These constraints help direct tests to hit a certain configuration more, modulate knob inputs to stay within a useful range or avoid generating an illegal test that is out of functional scope. To explain this using an analogy of the hardware design being a big dark box which has got bugs. To find those bugs, we have a strobe light, where a strobe is like running a test. We want to position the light optimally to use the available strobes (compute) efficiently. In CRT, we position the strobe light randomly, under constraints based on past knowledge.

Constrained Random (CRT) has significant scope for improvement as most of the randomly generated tests tend to pass. If there are a bunch of tests that have historically passed in almost all regressions, then simulating them again without an underlying design change affecting the input stimulus, is a waste of compute cycles (except when used to gather coverage data). For a verification engineer, it's hard to identify such test patterns and save compute on re-running them by adding constraints. ML can help do this automatically and frequently so that the learning stays relevant with the hardware design changes.

So far, we have introduced the baseline CRT setup and discussed where ML could be beneficial over the baseline. In the following section we define what is ML and then introduce our ML application.

B. *Machine Learning*

There are a few formal definitions of Machine Learning (ML) floating around, but the best one in my opinion that fits our problem ecosystem is "*The basic concept of machine learning in data science involves using statistical learning and optimization methods that let computers analyze datasets and identify patterns. Machine learning techniques leverage data mining to identify historic trends and inform future models."* [2]

In my opinion this definition sets the right expectation about the science of Machine Learning (ML). It should be the correct way for a human to make decisions backed by data. A very simple use-case of ML can be learning the equation of a line that fits historical points on a cartesian plane. Then we can use this line to predict y-value for a new x-coordinate. In Fig 1., the green dots are historical points and the line passing through them is a learnt model. By extrapolating the line using its equation we can predict the y-value for a new x-coordinate (red triangle).

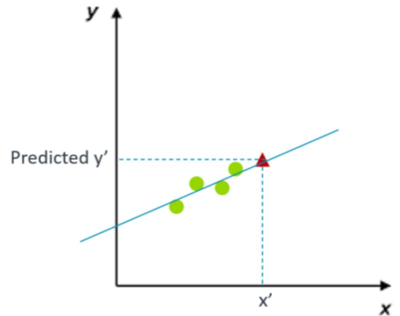

Fig. 1. Modelling a line to predict y' for a new input x'

C. Machine Learning (ML) for Verification

Now that we have defined ML, we believe it can be a useful technology to solve the simulation verification problems introduced at the end of sub-section (I-A). Using ML techniques, we build **classification models** which input the knob stimuli of a simulation test generated by a CRT generator. The ML model associates a **likelihood** of failure to every test in this regression. This likelihood is calculated by statistical analysis of tests, correlating their knob-value combinations to Fail/Pass outcome. So, a new test with knob-values that have had high correlation to the Fail decision historically, will have higher likelihood of failure. The generated tests are then **ranked** based on this likelihood and top k (where k is a user provided argument) are sent for simulation [3]. This helps us save compute.

For easy understanding, we now give an example using the strobe light analogy from sub-section (I-A).

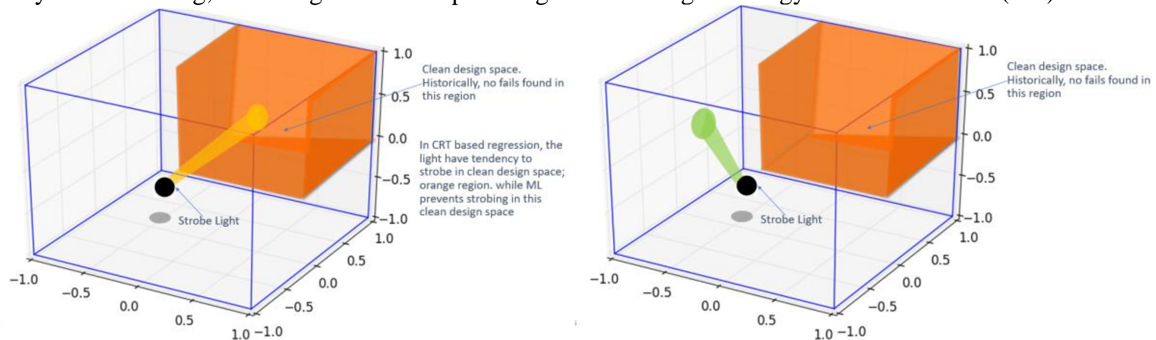

Fig 2. Strobe light analogy showing historically clean region highlighted in orange. No fails present in this region in past. On left, is a CRT regression that have no control over testing this clean design space. On right, ML prevents from using compute to test in this region.

In Fig 2., we claim that in past strobes, we have never found a failure in the orange region. More precisely, we are saying that in the history available to us, the likelihood of finding fails in the orange region is very small (approx. 0). With this knowledge our ML model learns to assign a very low probability of finding bugs to a light orientation in the orange region. Hence, even though a random position generator may generate orientations in that region, the ML model will prevent strobing in those orientations. A strobe is analogous to simulation cost and light positioning is analogous to test generation cost. Test simulation compute cost is greater than test generation cost, hence we save more than we spend. In this example we have cut down $1/8^{th}$ of the design space to test, using historical evidence.

In the following sections we discuss the ML details, architectural setup, and results of the work on a live project.

II. MACHINE LEARNING DESIGN AND IMPLEMENTATION

A typical Machine Learning design has two flows with composite action steps, illustrated in Fig 3. In this section we discuss each of the flow steps with our implementation details and modelling choices.:

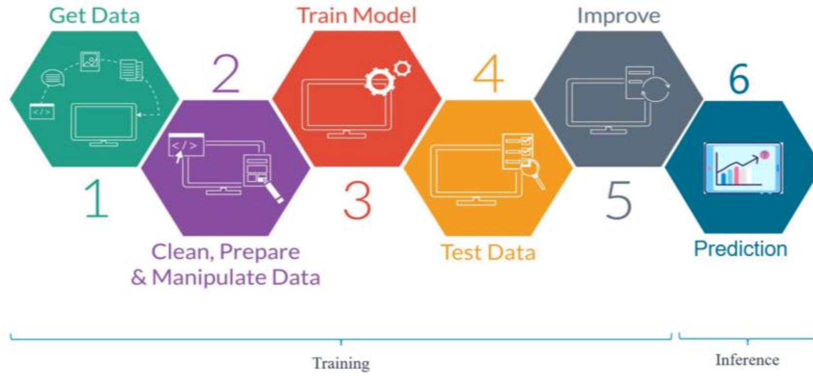

Fig 3. A typical Machine Learning design (Belsare, Karan & Goraya, Dr-Manwinder. (2022). Varıous Frameworks for IoT-Enabled Intellıgent Waste Management System Usıng ML for Smart Cıtıes. 10.1007/978-981-19-2069-1_55.)

A. Training

1) *Data Gathering*

Data Format: Simulation test data may exist in various forms. Sometimes it can be a command line with knobs and knob-values and sometimes it can be present in configuration files specifying stimulus randomizations. We can use our solution on either format with minor modifications in the data preparation step. [*The only restriction is to have the data accessible before test simulation. We can't use randomizations (knobs) evaluated during simulation, because if ML prediction happens during simulation, there is no way for the model to filter knob-values, mid-simulation.*]

Implementation: We use two weeks of the most recent simulation test data on one of our large IP designs, for ML training. We use this time window because it gives ML enough data points to train on. We don't use older data because hardware design is constantly changing with RTL feature development and bug fixes. Hence, simulation test data becomes stale in reflecting RTL behavior after some time. We must re-train periodically to keep the models relevant. We will discuss more about retraining in section III. Our data also has high degree of class imbalance with majority of historical tests passing against a very small number of fails.

In our verification setup, simulation test data is a command line of knobs controlling test stimulus. There are approximately 1000 knobs for 1 simulation test and 2 weeks of data results in 1 million simulation tests for training. Our typical training set after tokenization looks like Table 1:

Test #	*Knob-1*	*Knob-2*	*….*	*Knob-p*	*Fail*
Test_1	Val-1_1	Val-1_2	….	Val-1_p	True
Test_2	Val-2_1	Val-2_2	….	Val-2_p	False
:	:	:	….	:	:
Test_n	Val-n_1	Val-n_2	….	Val-n_p	False

Table 1: Simulation test data structure for training. n=1 million, p=1000

2) *Data Cleaning, preparation, and Preprocessing*

Since our data is a bunch of text command lines giving knob information about a test, we parse these command lines and tokenize the data into (knob, knob-value) pairs as illustrated in Table 1. The knobs act as columns (or feature names) and the knob-values are added to the respective test row. We can imagine that building a table like Table 1 will have missing values, variable datatype (int, float, objects, categorical, bool etc.) in columns. Hence, we preprocess the data to fill the missing values with defaults for every column, apply type conversion for mixed datatype columns, trim columns that have the same value in the entire dataset and normalize numerical data type columns. Machine ingestible data is usually numerical; hence there is also an encoding step involved that translates non-numerical data into encodings (e.g., One-Hot Encoding). Once all

these steps are completed, then we get a dataset ready for training. In our implementation, after the preprocessing steps we get a **normalized and encoded dataset** of 1 million rows and 900 columns roughly. This shape of the dataset varies with every re-training but sticks around similar values for a project. For more details on our preprocessing setup, please refer to [4]

3) *Model learning/ (Algorithmic) Training*
In this step we use statistical algorithms to fit an initial null hypothesis (naïve model) onto our training data by modifying its parameters; based on the error margins of the model from the training data points. This is an iterative step aiming to minimize the cumulative error margin. For model learning, we use different algorithms (for e.g., SGD [5], LGBM [6] etc.) and use the best estimator from Randomized Search based learning [7] to make predictions.
In a Randomized Search based learning, there are algorithm-specific hyperparameters that are randomly set in every training cycle. The sklearn implementation of this search takes an argument of how many times it needs to randomly initialize the hyperparameter grid. We have set it to 100. Both SGD and LGBM are Gradient Boosted decision trees, which are typically used in Supervised Learning problems. Another motivation to use Gradient Boosted trees is that they perform well on imbalanced data. An imbalanced dataset is where one class label is present in the majority. This is true in this case as Pass label dominates the dataset. Passing tests are in the high 90's % of the dataset. We tried training with other algorithms such as Logistic Regression, Decision Trees, and Extra-Tree Classifiers, but SGD and LGBM gave the best predictions on our dataset most often.

4) *Test Data/Cross Validation [8]*
Cross Validation is the feedback mechanism that drives the model to learn a generalized trend and not being overly tied to the training data set. ***Model training is an iterative process*** and not limited to one algorithmic execution on the entire training data. To cross validate we split the training data into a train set and a validation set. We first fit a model on the train set and then evaluate its predictive ability on the validation set. Since validation set is cut-out from historical data, we already have real value of the result for this set. As it was never fed to the model while training (just how future data would be unseen by the model), so it behaves like new data for model prediction. The model then makes predictions on this validation set which are compared to real results and model is scored. With every iteration, the algorithm optimizes this cross-validation score by varying model parameters.
For our implementation we use the Area under the ROC curve (AUC ROC [9]) score for cross validation. The split of the train set, and validation set is done 3 times to avoid bias towards one split. We use sequential cross validation strategy to create splits grouped by test dates. The model that performs best in predicting among all the splits is used (Best Mean AUC ROC) for prediction.

B. Prediction
Once we have learned the model, we use it to make predictions on new simulation tests. The verification testbench generates tests. Like training data, these tests comprise of knobs and knob-values as command line arguments. They are then passed through the exact data cleaning, preparation, and preprocessing step that were used while training. Now, the data is in an ingestible format for the model to make predictions. The model assigns a failure likelihood score to every test. This score is used to rank the tests. From this ranking, top k tests (where k is a user defined argument) are submitted for simulation.

In the following section we look at the combined architectural setup of this ML based regression with CRT based regressions in a simulation-verification setup.

III. Architectural Setup of Machine Learning for Verification

So far, we have gone through the steps to build an ML model which can make predictions about the likelihood of a test input to fail. Figure. 4 shows our architectural setup of running ML and CRT based regressions together on the same version of hardware design. **Our ML application is independent of the EDA tool used for simulation**.

Both approaches use a Constrained Random test generator that provides different test stimuli. In the ML based flow, stimulus first passes through ML model and gets ranked. Then the top-k tests are simulated, and the results are uploaded to a datastore. This data is then used for debugging failures, computing metrics, and model retraining.

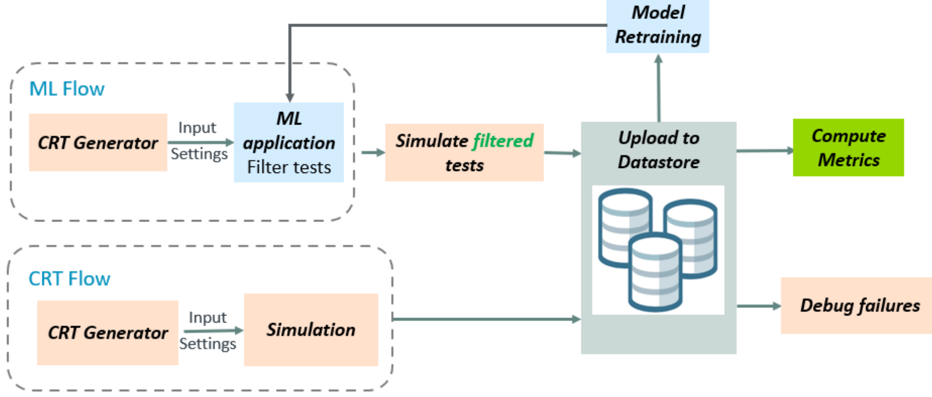

Figure. 4 ML and CRT Flow setup for Verification

We continue running CRT regressions along with ML based regressions for two important reasons:

a. Hardware design keeps changing with the introduction of RTL features and resolution of existing bugs. These changes can modify the behavior of the design to the same test stimulus, resulting in a different outcome (Fail/Pass). It means if a test has been historically passing every time, with a new update it can start failing. This change in the underlying behavior diminishes the model's predictive ability after some time and the model becomes stale/primitive. Hence, we need to periodically retrain the models to keep them relevant. Having secondary CRT regression generates new re-training data. Since design changes are very frequent (daily), to cover a significant modification to the design we tend to re-train every week using new data from previous week and old data from the week prior to that. This works in a sliding window fashion to capture recent changes and simultaneously model the most recent history.
b. Although ML should be the primary verification flow running greater volume of tests, it's wise to keep CRT as a secondary flow, because it may probe unexplored areas of the design. That's valuable as the current ML solution does not control test generation to drive tests towards the unexplored design spaces.

As part of our future work, there is a need to make the ratio of ML to CRT regression size adaptable to the phase of a project. We can imagine that there will be evolving phases of the project when running ML and CRT in close proportions would make sense and there will also be stable phases of the project when running one technique more than the other would be beneficial. It would depend on the objective of the verification team at a given phase of the project. If finding similar fails is the objective, running more ML might be beneficial, and to hit new/unexplored areas of the design more CRT, might be the right approach. In future we will be doing analysis of ML and CRT volumes over different phases of the project to come up with an adaptive technique to vary their ratios. It is not recommended to stop CRT regressions entirely, at any phase of the project for the 2 reasons mentioned above.

IV. EVALUATION METRICS

The value of using ML is realized by saving compute. So, ML must do the work done by CRT, but efficiently. The work here is to find failures. Failures are commonly bucketized into fail signatures. We assume that a unique failure signature (UFS) identifies a bug. With this understanding, we introduce two metrics denoting ***efficiency*** and ***quality*** of ML based approach against the baseline (CRT). We then estimate a ***cost*** metric in sub-section (IV-C)

A. Unique Fail Signature per CPU Hour (Efficiency)

We use this metric to determine the efficiency of the ML and CRT approach. We normalize the number of UFS found by the number of CPU Hours spent to find them. In a production setup, ML will run smaller size regressions compared to CRT. This normalization ensures a fair comparison between the two approaches.

$$Efficiency\,(UFSperCPUHour) = \left| \frac{COUNT(UFS)}{\sum_{CPUHours}} \right|_{Month} \tag{1}$$

The approach aggregates data for regressions over a month getting UFS count for fail signatures that appeared in that month. The denominator is summing the CPU Hours used by the regressions to simulate tests over that month.

Table 2. An illustrative example of the formula above.

Flow	Machine Learning Flow	CRT Flow
UFS count for the respective flow	*x*	*y*
Sum of CPU Hours used by the respective flow simulation	*M*	*N*

Table 2. presents an example to depict our formula in practice. The Efficiencies will be defined as:

$$Efficiency\,(UFSperCPUHour_{ML})) = \left|\frac{x}{M}\right|_{Month} \qquad Efficiency\,(UFSperCPUHour_{CRT})) = \left|\frac{y}{N}\right|_{Month}$$

B. UFS Recovery Rate (Quality)

We expect ML to find fails more efficiently than CRT (as demonstrated in the Results section below). It would be wise to see what portion of the fails present in the design are picked by either approach, normalized by the volume of compute each uses. This touches the quality aspect of both the approaches. In other words, if we have 1 CPU Hour available, then using that, what portion of the existing UFS can each flow find. This calculation assumes that the UFS found by ML and CRT together are the total number of observable UFS present in the hardware design. We aggregate the results by month.

$$Recall(ML) = \left|\frac{COUNT(UFS_{ML})}{COUNT(UFS_{ML\cup CRT})}\right|_{Month} \tag{2}$$

$$UFSRecoveryRate(ML) = \left|\frac{Recall(ML)}{\sum_{CPUHours(ML)}}\right|_{Month} \tag{3}$$

Reusing Table 2, the UFS Recovery rate for both the flows will be:

$$UFSRecoveryRate_{ML} = \left|\frac{\frac{x}{x\cup y}}{M}\right|_{Month} \qquad UFSRecoveryRate_{CRT} = \left|\frac{\frac{y}{x\cup y}}{N}\right|_{Month}$$

C. Cost Savings

The above two metrics are comparative results between ML and CRT. But as recommended by our architecture in section III, we intend to save compute costs when ML and CRT are run together. Hence, we tried estimating the compute savings ML can have in this experimental setup. We use data from the same time in section (IV-A.) and (IV-B.). The monthly inverse of UFS/CPU_Hour from (IV-A) is the rate of finding UFS (i.e., CPU_Hours/UFS). So, in a pre-ML world CRT would have the responsibility to find all the UFS's.

UFS found by both ML and CRT are called common fails and must be credited to both the techniques fairly. The problem to split credit of common fails is a complicated one, which involves analysis of the failure curve for both the techniques and their different regression volumes. This is part of our future work and hence we used a few different approaches to share the common fails. We tried assigning all common fails to CRT but that is a flawed approximation discrediting ML. So, we shared the credit of finding the common fails half and half between the two approaches. By this approximation, CRT would have to find the additional UFS's – i.e., the UFS found only by ML and half of the common UFS's, to find the existing UFS's alone. Multiplying the count of the additional UFS to CRTs' rate of finding UFS would give the additional CPU Hours that CRT would spend to find the additional UFS by itself versus ML's ability to find the same number of additional UFS.

$$UFSFindRate(CRT) = \frac{1}{Efficiency\,(UFSperCPUHour)_{CRT}} \tag{4}$$

$$Common_UFS = COUNT(UFS_{CRT}) + COUNT(UFS_{ML}) - COUNT(UFS_{ML\cup CRT}) \tag{5}$$

$$AdditionalUFS = COUNT(UFS_{ML\cup CRT}) - COUNT(UFS_{CRT}) + \frac{1}{2}[Common_UFS] \tag{6}$$

$$Additional_CRT_CPUHours = AdditionalUFS * UFSFindRate(CRT) \tag{7}$$

$$Additional_ML_CPUHours = AdditionalUFS * UFSFindRate(ML) \tag{8}$$

$$Compute_Savings = (Additional_CRT_CPU_Hours - Additional_ML_CPU_Hours) \tag{9}$$

Using Table 2, our savings will look like this:

$$Compute_Savings = \frac{1}{2}((x \cup y) - y + x)\left[\frac{N}{y} - \frac{M}{x}\right]$$

V. RESULTS

In this section we look at the results of the metrics in section IV, on a live design project over the entire duration of the project involving all the hardware development lifecycle phases. The results are aggregated by every month:

A. Unique Fail Signature per CPU Hour

Figure 5. shows monthly aggregation of our efficiency metric. Over the 15 months of project data collected, we observed ML is more efficient than CRT with an average improvement of **1.35x**.

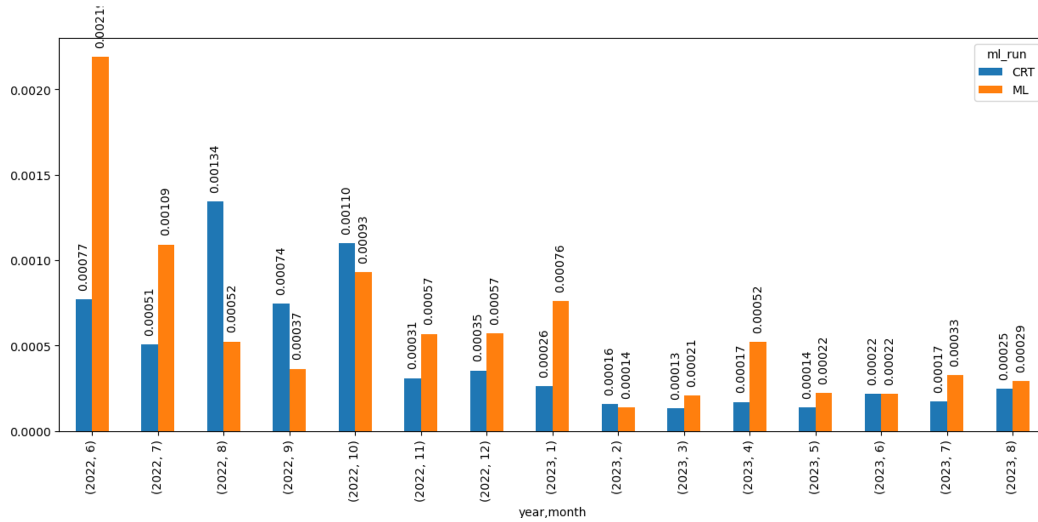

Figure. 5 UFS/CPU Hour values for ML and CRT from June 2022 to September 2023

B. UFS Recovery Rate

Figure 6 Shows the ML and CRT Quality depiction over the same period. We see that ML is doing better than CRT baseline averaging to **1.6x**

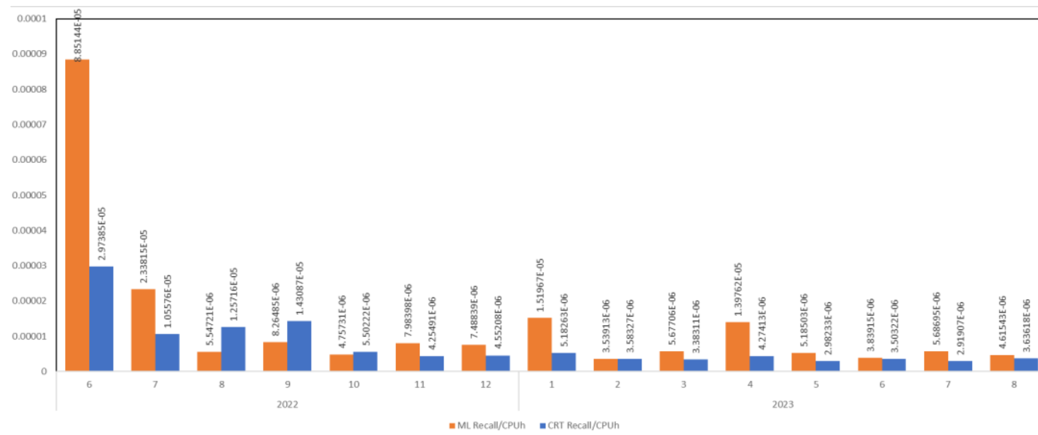

Fig 6 UFS Recovery Rate for ML and CRT form June 2022 to September 2023

There are months where CRT performed better in metric (A) and (B). That is possible due to model primitivity.

C. Cost Savings

On cumulating the monthly compute savings over the duration of the same 15 months, we attain an estimated **7.6%** of compute cost saving when using ML with CRT instead of just CRT. For calculating the percentage savings we normalize the original compute time for both ML and CRT to account for their own UFS's. We analyzed our ML model training compute costs to account for in this metric, which was negligible compared to the simulation costs.

The ML test volume is currently 16% of the CRT volume and in future the cost savings can be improved by

a) *Increasing the proportion of ML tests with respect to CRT*. That being said, given the saturation based nature of fails in a hardware design; this volume needs to be adjusted to a breakeven point to stay profitable. Running ML beyond the point of fail saturation will increase its CPU_Hours rapidly but not the fails as much. Hence, we need to find that saturation point to maximize the benefits of ML's better efficiency at finding fails. Finding the saturation point is not a trivial problem because the hardware design keeps changing with every regression and hence the fail saturation curve also changes everytime.

b) *Improving the ML model*. A better performing model will improve ML technique's efficiency and will hit the UFS saturation point in lesser number of CPU_Hours.

Such deployment parameter tuning and improvements are mentioned in the Future Work section (VII)

VI. Challenges

1) A common challenge in ML is class imbalance. It's also true for our use case. The number of fails and passes present in our training data are highly imbalanced (5:95). Hence, we use gradient boosted algorithms that handle class imbalances well.
2) ML development has to happen during live projects. Sparing additional compute for ML regressions is a significant investment. We carefully set ML deployment test volumes to not exploit critical project resources.
3) The expected behavior of ML must be adaptable to the phase of the project. Some phases demand reproduction of old failures and some demand finding new failure configurations.
4) ML is a science of probability and hence comes with a margin of approximation. It's hard to justify and explain the reason for ML picking a test stimulus as doing it is a lengthy process.

VII. Future Work

As we have seen, there exists imminent value in changing simulation-based verification strategy from CRT only, to ML + CRT. But this raises interesting deployment questions, and our future work is to answer these questions based on experiments and data analysis. Some of this work must happen frequently to adapt to every phase of a project:

Deployment Improvements:

1. *What should be the ratio of ML to CRT tests for best savings in every phase of the project?*
   We did early experiments to set the ML:CRT test ratios. To get reasonable value from ML, without disturbing the project verification strategy drastically, we fixed the ML volume at 16% to that of CRT. As we now have more deployment data, we will conduct experiments to push the ML:CRT ratio further.
2. *What should be the ratio of generated test for ML, to the number of filtered tests?*
   As mentioned in earlier sections, ML filters on a CRT generated test set. We have analyzed and established that test generation compute is negligible to test simulation compute. With that in mind, it is wise to give ML a large set of generated tests to make predictions on, hence exploring a larger design space while running limited simulations. From past analysis, we arrived at a decision for ML to pick 12% of the generated tests. This analysis must be rolling to adapt to the various stages of a project.

Machine Learning Improvements:

3. *We can tune our models more to achieve better predictions. We are doing this by feature selection, use of other training algorithms, creation of better Cross Validation scoring functions etc.*
4. *We can extend this application of ML, to find efficient test set for quantifying RTL coverage.*
5. *We will use generative AI to control knob generation where ML generates optimal knob values to hit fails.*

VIII. Conclusion And Discussions

We have established that using ML we can automate the process of filtering simulation tests to obtain an efficient test regression by a factor of **1.35x, with an improved UFS Recovery rate averaging 1.6x better than CRT**. This can help us run smaller size regressions and use the saved compute to reduce verification costs or run additional verification regressions, potentially resulting in identification of more bugs. In the same project, running ML at 16% ratio against CRT saved us **7.6%** of the total compute costs. This is a significant saving given Simulation Verification can be the major consumer of budget in a hardware design project.

Acknowledgment

I would like to acknowledge my employer's ML & CPU team for development, budgets, and review of this paper.

References

[1] Constrained Random Verification, Yuan J.; Pixley, C.; Aziz, A.; 2006, XII, 254p. 72 Illus., Hardcover, ISBN: 978-0-387-25974-5

[2] https://ischoolonline.berkeley.edu/blog/what-is-machine-learning/

[3] "Case study: real-world machine learning application for hardware failure detection", H. Shin; Proc. 18th Python in Science Conf. (SciPy), pp. 5-12, 2019.

[4] H. Shin, "Data-Centric Machine Learning Pipeline for Hardware Verification," 2022 IEEE 35th International System-on-Chip Conference (SOCC), Belfast, United Kingdom, 2022, pp. 1-2, doi: 10.1109/SOCC56010.2022.9908095.

[5] Sebastian Ruder: "An overview of gradient descent optimization algorithms", 2016; arXiv:1609.04747.

[6] LightGBM: A Highly Efficient Gradient Boosting Decision Tree, Guolin Ke et al.; Advances in Neural Information Processing Systems 30, 3146–3154 (2017)

[7] Petro Liashchynskyi, et al.: "Grid Search, Random Search, Genetic Algorithm: A Big Comparison for NAS", 2019; arXiv:1912.06059.

[8] Refaeilzadeh, P., Tang, L., Liu, H. (2009). Cross-Validation. In: LIU, L., ÖZSU, M.T. (eds) Encyclopedia of Database Systems. Springer, Boston, MA. https://doi.org/10.1007/978-0-387-39940-9_565

[9] A. P. Bradley, The use of the area under the roc curve in the evaluation of machine learning algorithms, Pattern recognition 30 (7) (1997) 1145–1159.