Case study: Real-world machine learning application for hardware failure detection

Hongsup Shin[‡*]

Abstract—When designing microprocessors, engineers must verify whether the proposed design, defined in hardware description language, does what is intended. During this verification process, engineers run simulation tests and can fix bugs if tests have failed. Due to the complexity of the design, the baseline approach is to provide random stimuli to verify random parts of the design. However, this method is time-consuming and redundant especially when the design becomes mature and thus failure rate is low. To increase efficiency and detect failures faster, we can build machine learning models by using previously run tests, and address the likelihood of failure of new tests. This way, instead of running random tests agnostically, engineers use the model prediction on a new set of tests and run a subset of tests (i.e., "filtering" the tests) that are more likely to fail. Due to the severe imbalance (i.e., >99% test success and <1% failure), I trained an ensemble of supervised (classification) and unsupervised models and used the union of the prediction from both models to catch more failures. The tool has been deployed as a complementary workflow early this year, which does not interfere the existing workflow. After the deployment, I found that the the "filtering" approach has limitations due to the randomness in test generation. In addition to introducing the relatively new data-driven approach in hardware design verification, this study also discusses the details of post-deployment evaluation such as retraining, and working around real-world constraints, which are sometimes not discussed in machine learning and data science research.

Index Terms—hardware verification, machine learning, outlier detection, deployment, retraining, model evaluation

Introduction

Simulation-based hardware verification

Hardware verification is the process of checking that a given design correctly implements the specifications, which is the technical description of the computer's components and capabilities. It is recognized as the largest task in silicon development and as such has the biggest impact on the key business drivers of quality, schedule and cost. In the computer hardware design cycle, microprocessor manufacturing companies often spend 60-70% of the cycle dedicated to the verification procedure. Traditionally, two techniques have been used: formal and simulation-based (random-constraint) methods [Ioa12]. The former adopts a mathematical approach such as theorem proving and requirement checks [Wil05], which provides exhaustiveness but doesn't scale well with design complexity. Due to the exponentially-growing design complexity, the more widely used approach is the simulation-based testing, which simulates a design (i.e., each line in hardware description language) by providing stimuli to tests. During simulation-based testing, engineers provide a set of constraints to stimuli so that they can direct tests to a certain direction. However, it is never possible to target certain design parts deterministically and engineers often depend on previous knowledge or intuition.

Failures (bugs) in hardware verification

Hardware verification can be compared to unit testing in software engineering, especially since design functionalities are realized in hardware description language (HDL) like Verilog. Similar to software testing, hardware verification process involves checking whether simulations of the code written in HDL with a set of given input values (i.e., tests with certain inputs), show desirable behavior. If a test returns undesirable output, it is considered as a failure (bug). To fix the failures, engineers modify the HDL source code such as by fixing "assign" statements or by correcting or adding conditions (e.g., "if" statements), and so on [Sud08]. This HDL-level hardware verification is one of the many steps in hardware testing, occurring before physical design is implemented. This low-level verification is a critical step in hardware testing because fixing a bug in a higher level (e.g., in physical design or even in a product) is more costly and challenging.

Previous machine-learning based approach

The ultimate goal of hardware verification is to have a (close-to) failure-free design. From the simulation-based testing perspective, this is an exploration problem where machine learning can be useful. For instance, reinforcement learning algorithms can be used to explore a complex parameter space by learning a reward function [Ioa12]. However, this approach is not feasible because the simulation-based testing is non-deterministic and intractable, which makes it difficult to estimate the level of stochasticity. This is mainly because the motivation for the simulation-based approach is randomization, which often occurs in multiple steps (i.e., a value in an input setting randomizes a value in the next step, which also randomizes a value in the following step, etc.). The testing tools have been built to often ignore tracking of these setting values and the information on probability distributions used in the randomization process were left out. To address this, a few studies [Bar08], [Fin09] adopted probabilistic approach but they failed to mention actual implementation in production cycle and scalability issue. The majority of the previous research on hardware verification with the simulation-based testing approach

* Corresponding author: hongsup.shin@arm.com
‡ Arm Research

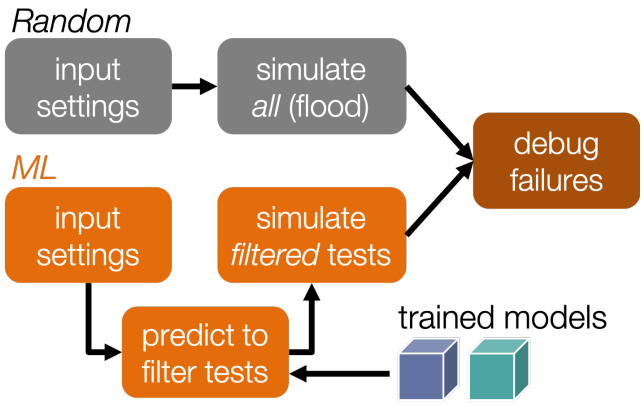

Fig. 1: *Overview of the prototype pipeline. Top: the existing workflow (randomized testing). Bottom: the complementary machine learning (ML) flow. In the final deployed version, approximately 1000 test candidates are provided to the ML flow, which passed about 400 tests. This corresponds to the 10% of the number of the tests in the top flow. The cubes correspond to the pre-trained machine learning models (blue: a supervised model, green: an unsupervised model).*

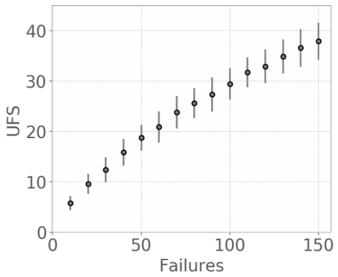

Fig. 2: *Relationship between the number of failures (x axis) and the number of unique fail signatures (UFS) on the y axis (mean and standard error). To generate the error bar, I ran 100 simulations where in each simulation, I draw N failed tests among a pool of 250k tests and counted the number of UFs. The more failures occur, the more UFS are found.*

has focused on supervised learning [Mam16], [Bar08], [Wag07] and evolutionary algorithms [Ber13], [Cru13].

Simulation-based testing in practice

In practice, engineers build a testbench to house all the components that are needed for the verification process: test generator, interface, driver, monitor, model, and scoreboard. To run tests, verification engineers define a set of values as *input settings*, which can be compared to input arguments to a function. These values are passed to the test generator, and under certain constraints, a series of subsequent values that stimulate various parts of the design are *randomly generated*. This information is then passed to the interface through the driver. The interface interacts with a design part (register-transfer level (RTL) design written in HDL) and then the returned output is fed into the monitor. To evaluate the result, the desirable output should be retrieved. This information is stored in the model, which is connected to the driver. A test is identified as failure when the the desirable output from the driver (through the model) and the output from the monitor do not match. In addition to the binary label of pass or failure, we also obtain a log file of failure, if the test has failed. This log file contains detailed information of the failure. Each failure log is encoded to a 8-digit hexadecimal code by a hash function. This code is called *unique failure signature (UFS)*. Instead of inspecting every failure log, in general, engineers are more interested in maximizing the number of UFS that are collected after a batch of tests because gathering a large number of UFS means they have found failures with a great variety.

Random generation of the test settings in the test generator is intended for running a batch of tests automatically almost daily to explore random parts of the design efficiently. Once engineers run tests with certain input constraints, *settings*, and the simulation is finished, the results are obtained. The way that engineers control the input settings vary widely. In an extreme case, they only control the seed number of a pseudo-random number generator in the test generator for the entire set of the input settings. Normally for a test, engineers have a set of input settings, which either turn on and off of a setting or controls stochastic behavior a setting by defining what kind of values the setting can take. For instance, if a certain input setting has a string value of *"1-5"*, it indicates that the final stimulus value can be *any integer from 1 to 5*. As above-mentioned, testbenches do not track any information such as which value has ended up chosen eventually. Hence, it is extremely challenging to guide a testbench to generate a specific value of the input settings. This is why building a machine learning model is challenging because two tests with the exact same values of an input setting can result in two different outcomes. Additionally, engineers make changes to the design almost every day, which include a new implementation or modification in the design or bug fixes. This affects the test behavior and in turn, data generation process, which implies, the data distribution can potentially change almost daily (i.e., frequent data drift).

Working around the stochastic test generation

This situation requires a unique approach. It is impossible to eliminate randomness in the test generation step, which makes it difficult to guide testbench to test specific input values or parts of the system. Instead, we leave the inputs to be generated randomly and filter them afterward. By using the labeled data from previous tests (i.e., tests that were already simulated), we build a machine learning model (classifier) that predicts whether a test will fail or pass with a given set of input settings. Then we provide a large set of test candidates (a number of tests with random input setting values) to the trained model, which can tell whether a test will fail or not. Using the prediction, we only run a subset of tests that are flagged as failure, instead of running the entire test candidates agnostically. This can bring cluster savings and make the verification process more efficient. However, the existing simulation-based testing with random constraints *should remain* because we still have to explore new design parts, which in turn provide new training data for model update. Hence, we propose two parallel pathways (Fig. 1); one with the default randomized testing and the other where an additional set of test candidates are provided and then failure-prone tests are filtered and run. This way, we can continue collecting novel data from the first pathway to explore a new input space while utilizing the data from previous tests.

Post-deployment analysis

I used both supervised and unsupervised models to address the severe class imbalance problem and used the union of the prediction from both models. With this approach, for a set of independent

Model candidates	Recall	Efficiency
#1	0.70	1.25
#2 (chosen)	0.66	1.85
#3	0.85	0.55
#4	0.25	2.50

TABLE 1: *Example of model candidate scores and how the best model is chosen. In the tuning process, both recall and efficiency are considered. Efficiency of 1 means the ML flow is as efficient as the random flow. This becomes the lower bound of model performance. #3 is ruled out because even though it has the highest recall, the efficiency is lower than 1 (baseline). Then, #1 is the model with the highest recall. However, instead of choosing this, I look at other candidates with in a margin (0.05 in this case) from the maximum value of the recall, meaning all the candidates that have recall values between 0.70 (maximum) and 0.65 (=0.70-0.05). In this example, #2 has higher efficiency than #1 and is within the margin. Hence, #2 is chosen as the best model.*

testing datasets, it was possible to find 80% of unique failure signatures (Fig. 3) by running only 40% of tests on average, compared to running tests based on the original simulation-based method. The tool has been deployed in production since early this year in our internal cluster as a part of daily verification workflow, which is used by verification engineers in the production team. It is not common in both machine learning and hardware verification literature to find how suggested models perform in real-world setting; often machine learning papers show performance based on a single limited dataset or use commonly used benchmark datasets. In this paper, I address this and attempt to provide practical insights to the post-deployment process such as decisions regarding the automation of model retraining and addressing randomness in post-deployment datasets.

Methods

Data

Simulation-based testing is run almost every day via testbench. Every simulated test and the outcome (i.e., test success (pass or failure) and UFS if a test has failed) is stored in a database. To address the issue of data drift over time, we collected two datasets. The first dataset ("snapshot") was generated from a same version of testbench (115k tests). For the second set, we collected a month's worth of data (ca. 6k tests per day). The second dataset ("1-month") is collected specifically to simulate retraining scenarios and to challenge our model for every-day changes in the testbench (150k). Both datasets are from a specific unit of a microprocessor with a specific test scenario. The input dataset has individual tests as rows and test settings (stimuli) as columns. These settings are specified by verification engineers. The total number of settings are in the range of several hundreds. The output dataset has tests as rows and two columns, one for pass/failure binary label and the other for UFS for the failed tests.

Data preprocessing

The input data was preprocessed based on the domain knowledge of the verification engineers. In the raw data, roughly 70% of the data was missing. This is because when the value of an input setting in a test is the same as a default value of the setting, it was not specified by the engineers. Using software analogy,

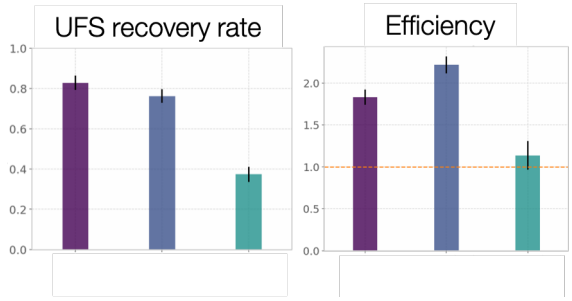

Fig. 3: *Unique failure signature (UFS) recovery rate (left) and efficiency (right) metrics across 15-day (1 month) performance for the three models (union, supervised and unsupervised). The dashed orange line in the efficiency plot shows average fail-discovery rate (the lower bound of the efficiency metric). Note that the union approach catches more UFS but lowers efficiency because more tests should be run.*

this is similar to not having to specify an input argument value in a function if it already has a default value for that argument. The engineers were able to obtain the default values, which fixed the missing data issue. There were about 20% object (i.e., non-numerical) columns. Some of them were nominal columns (e.g., "name1", "name2") but the majority turned out to be numerical values in quotes (e.g., "5", "100"), quoted ranges (e.g., "1-5", "50-100") or a dictionary with key-value pairs in quotes. For the quoted numerical values, I simply stripped the quotes and converted them to float. For the quoted ranges, it was not straightforward because these columns have uncertainty information in them. For instance, "1-5" means any values from 1 to 5 and there was no information about the probability distribution. Although I initially considered treating them as nominal, I decided to take the mean of the minimum and maximum values of a range value: for "1-5", it would be represented as (1+5)/2 = 3. This way, it might be possible to preserve some numerical information about the range in the input data. For the quoted dictionary, I parsed them and expanded to multiple columns so that each key represents a column. Finally, I dropped columns that are non-informative (i.e., single unique value) and duplicates. This results in about 10% increase of the number of columns, which was still in the range of several hundreds. The output datasets did not require preprocessing.

Models

I used an ensemble of a supervised and an unsupervised learning model. Due to the severe class imbalance between passes and failures (near 99% pass and 1% failure ratio) in the training data, we can either train a supervised model with adjusted class weight or train an unsupervised model to detects outliers (i.e. failures). For the unsupervised, because the majority of our training data is passed tests, it is possible to consider the failures as outliers or abnormalities. In a preliminary analysis, I found that the supervised and the unsupervised models provided predictions that are qualitatively different. The unique failure signatures (UFS) from the supervised model's and the unsupervised one's predictions were not identical although there were some overlaps. Thus, when we computed the union of both predictions, we did see a small increase of UFS recovery across many testing datasets. Due to the frequent changes in data generation process, I decided to use algorithms robust to frequent retraining and tuning (i.e., faster training time). We used a group of non-neural-net scikit-learn

(v0.20.2) classifiers as supervised and isolation forest as unsupervised learning algorithms. For both cases, I conducted randomized search to tune the hyperparameters and select the best model. For the supervised, I used algorithms such as logistic regression and tree-based ensemble methods (random forest, gradient boosting, extra trees). The winning algorithm was the logistic regression with L2 regularization, potentially because the preprocessed input data has high sparsity (more than 50%).

Engineers care more about the unique failure signatures than simple binary labels. When a number of failures are found in test simulation, if the majority have the same failure signatures, it means we found failure that are very similar to each other, which has little value to the engineers. Hence, it would make sense to have an objective function that maximizes the number of UFS found, for instance, by formulating the problem as multi-class classification where each class corresponds to a failure signature. In the training data, each failure signature is found mostly just once or a few times, which makes it difficult to use in model training. However, I found that the number of failure signatures increases with the number of failures (Fig. 2); the more failures we find, the more unique failure signatures are retrieved. This suggests that as long as the binary classifier does a good job catching failures, it is likely to increase the number of unique failure signatures.

Metrics and hyperparameter tuning

For both supervised and unsupervised models, I used recall and precision as basic metrics (for model selection) but also used more practical metrics to increase interpretability and address the number of unique failure signatures, which engineers care about. I defined the following two metrics: *unique failure signature (UFS) recovery rate* and *efficiency*.

$$\text{UFS recovery rate} = \frac{\text{card}(S_{\hat{y}=1})}{\text{card}(S_{y=1})},$$

where S is a set of UFS, y and $\hat{y}$ are true and predicted labels of failure (0 as pass and 1 as failure), and $\text{card}(S)$ is the cardinality of the set, S, also known as the unique count of the set. Hence, $\text{card}(S_{\hat{y}=1})$ means the number of the UFS in the tests that are predicted as failure and $\text{card}(S_{y=1})$ as the total number of UFS in all failed tests in training data. This metric is similar to recall but here we focus on the retrieval of UFS instead of the binary labels.

$$\text{Efficiency} = \frac{\text{Precision}}{\frac{\sum_{i=1}^{N}[y_i=1]}{N}},$$

where N is the total number of the tests in the training data. In the deployment setting where we run both the default and ML flows, N is the total number of the tests in the *default* flow. Efficiency is defined to easily understand how efficient the ML flow is compared to the baseline. The numerator is the precision of a ML model and the denominator is proportion of the failures in training data (or the default flow), which means how often we find failures on average when running randomized tests (i.e., average fail-discovery rate). This metric can be used as a lower bound of model performance. Due to the trade-off between recall and precision, attempts to maximize recall will decrease precision. However, we do not want the precision to be lower than the average fail-discovery rate, because otherwise, the randomized testing would be enough or even better than the ML flow. Therefore, ideal model performance should show efficiency score larger than 1.

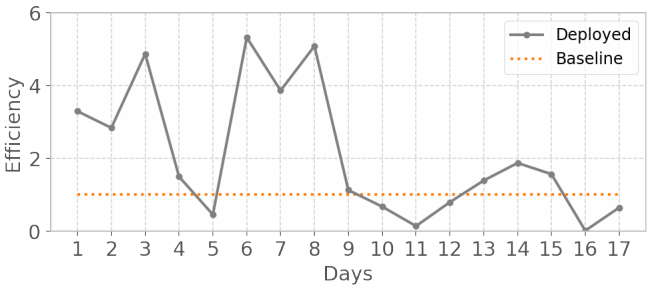

Fig. 4: *First 17 days (3k-4k tests per day) of model performance (efficiency) after deployment. The performance fluctuates widely (all the way up to more than 5 then sometimes plummet to zero). Note that the models have not been retrained during this period.*

Since the efficiency metric provides lower bound to model performance, when tuning the hyperparameters, instead of looking at the combination with best recall, I use the following rule to select the best among the model candidates. First, the candidates with efficiency smaller than 1 are dropped. Then the maximum of the recall values from the rest of the candidates is identified. Instead of selecting the candidate with the highest recall, i set up a margin (0.05) from the maximum recall and check whether there are candidates that are within the margin. Among these candidates, I chose the one with the highest efficiency. This way, without compromising the recall too much (only the margin), the model with higher efficiency can be chosen. The example is shown in Tab. 1.

Results

For the *snapshot* dataset, the testing data (50% holdout data in 10 sets; each set is generated independently via the testbench) shows that the union predictions from the trained supervised and unsupervised models achieved 82 ± 2 % (mean $\pm$ sem) UFS recovery rate and efficiency of 1.8 ± 0.1 (mean $\pm$ sem). Similar results were obtained in the *1-month* dataset (Fig. 3). Note that in the figure, UFS recovery rate increased when we combine the predictions from the supervised and unsupervised models but efficiency is lower because the union model requires running more tests. As a sanity check, since precision score was low (due to class imbalance), I ran a permutation test (100 runs) and found the model performance was significantly different from the permuted runs ($p = 0.010$). Overall, in both datasets, on average, the union approach flagged about 40% of the tests. This suggests, we can find approximately 80% of UFS by only running 40% tests compared to the existing random flow.

Post-deployment analysis

Deployment

Other engineers and I wrote a Python script within my group, which is a command-line tool that engineers can run without changing their main *random* flow. The script takes test candidates as input and by using the pre-trained models, make a binary prediction on whether a test candidate will fail or not. Note that whenever new test candidates are provided, we run a separate script that preprocesses the new data to be ready to be consumed by the pre-trained models. The test candidates are randomly generated by using the testbench and normally we generated about 1k test candidates so that at the end about 400 tests are

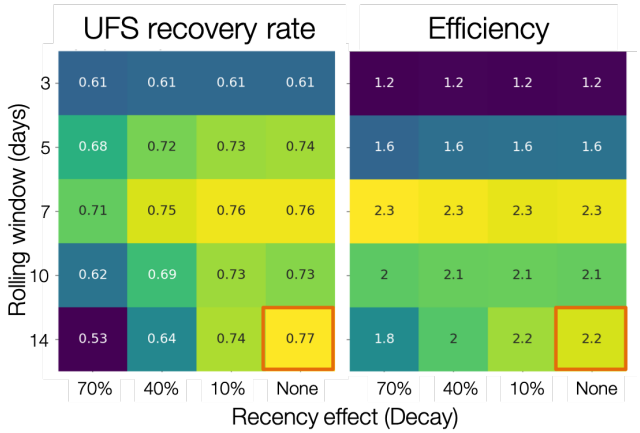

Fig. 5: *Average model performance metrics obtained by simulating various retraining scenarios. The x axis shows decay parameter (larger values mean faster decay), which decide the weights applied to training data. The y axis shows rolling window in the number of days, which decides training data size. For both top and bottom plots, brighter colors are more desirable. The marked orange squares show the final decision on training (i.e., 14-day window without decay)*

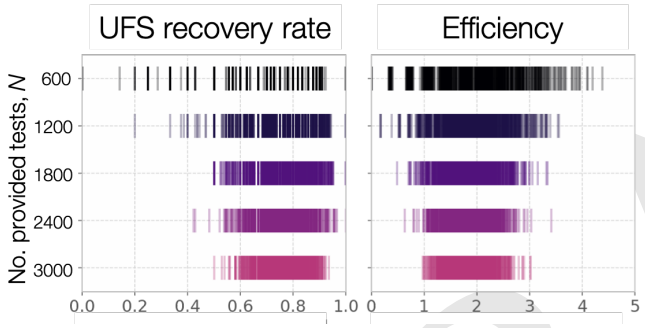

Fig. 6: *The effect of the number of tests that are provided to the models and the performance variability. Each vertical line represents a single simulated run. Since we use the models to filter out the test candidates, the fewer tests we provide to the models, more likely that performance depends on how good the initial test candidates are. The more tests we provide, the less variable the performance becomes.*

filtered, which is the upper limit of the number of additional tests we can run. We decided to adjust the number of tests as we have better assessment of the model performance after the deployment. Finally, the script returns the unique identifier of the test candidates that are flagged as failure by the models. Then the script invokes a testbench simulation where it runs the filtered tests. After the deployment, we found that model performance had high variability. Figure 4 shows the model performance of the first 17 days (no retraining). The efficiency values were often larger than 1 but sometimes they changed dramatically. In the following sections, I will address how I attempted to resolve this issue and found caveats of the "filtering" approach.

Data for retraining

During the initial deployment stage, we retrained the models manually whenever we made major changes in the tool, for instance how we preprocess data or whenever the production engineers announced that there was a major change in the testbench or the design. In order to decide how much training data we would use to optimize the performance, we conducted an experiment by varying the size and the weight of the training data. Theoretically, it's possible to use the entire suite of tests that were every run. However, this requires long training time and it's possible that very old test data would be useless if the design has changed a lot since then. Hence, in the experiment, we implemented a varying size of rolling window and weight decay. The rolling window size decides the number of N consecutive days to look back to build a training dataset. For instance, if $N = 7$, we use the past 7 days worth of simulated tests as our training data. The weight decay takes into account the recency effect of changes in the testbench; the data that was generated more recently has higher significance in training. We used 5 different windows ($N = 3, 5, 7, 10, 14$) and multiplicative power decay with various power parameters to compute the weight w, ($w(t) = x^t$ where x is the power parameter (0.3, 0.6, 0.9, 1 (=no decay)) and t is the number of days counting from today). For instance, if $x = 0.9$, tests that were run 2 days before today are 10% less important than yesterday's tests. These weights are applied to the objective function during training by using `sample_weight` parameter in scikit-learn models' `fit()` function, which allows users to assign weights during model fitting for every single data point. Since multiple tests are generated for each day, they each get the same weights. Note that this weight adjustment was added on top of the class weight adjustment (`class_weight='balanced'`).

All combinatorial scenarios were tested via simulation across multiple datasets (Fig. 5). When the rolling window was too small (e.g., $N = 3$), performance was low in both UFS recovery and efficiency metrics, which suggests 3-day dataset might not be enough for training. Having more dramatic decay tends to mimic the effect of having a smaller rolling window and generally degraded performance. In terms of performance stability over time, naturally, having a longer rolling window seemed better. As showed in Fig. 5 as orange box, we decided to use 14-day window without any decay even though the efficiency value was slightly higher in 7-day without any decay. This was to consider the fact that we might have to run a smaller number of tests in the future and thus 7-day window might not provide enough tests for training.

Random-draw effect

It is suspected that the fluctuation in performance (Fig. 4) might have originated from the fact that we provide a set of test candidates and let the model filter them out. This means, the quality of the test candidates we provide can decide the model performance. This is particularly important because the test candidates are generally randomly in the testbench. It is possible that by chance, the candidates we provided on a day might be more challenging to the models, which may result in low performance. I simulated the effect of random draw by varying the number of tests that we provide to the models (Fig. 6). I found that the more tests we provide, the more stable model performance becomes for both UFS recovery rate and efficiency. We have been providing about 1000 tests to our deployed tool (somewhere between the first and second at the top in the raster plots in Fig. 6) and it is very much possible that efficiency can be lower than 1 in that case. For the simulation in Fig. 6, we used a pool of 25k tests. Considering the fact that the actual number of possible tests we can every generate is much more than 25k, the variability in performance in reality could be more severe.

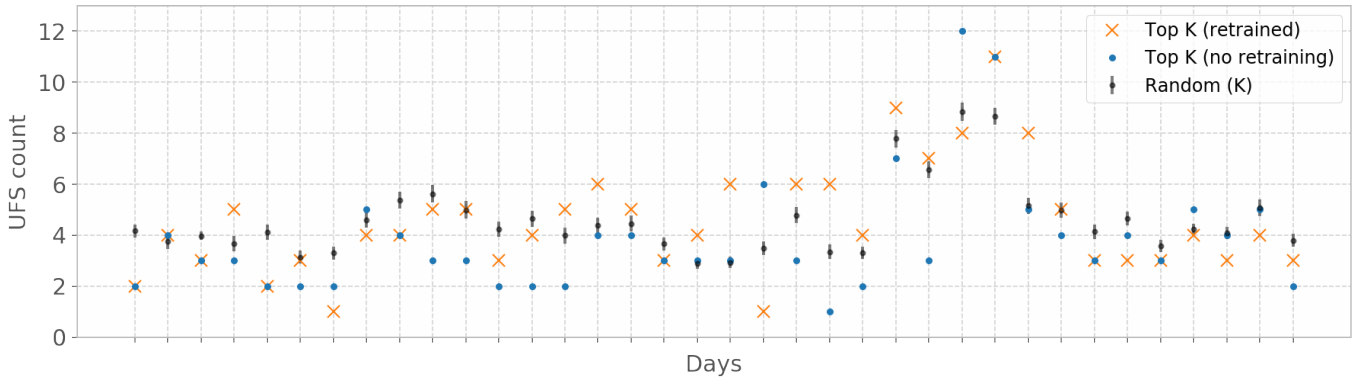

Fig. 7: *Comparison between randomly drawn K tests and model-filtered K tests (K=400) for 36 days after deployment in terms of the number of unique failure signatures (UFS). Prediction probability and anomaly score were used to rank the filtered test candidates and choose the top K tests to run (the orange crosses and blue dots). For the orange crosses, the models were retrained and tuned whenever the model performance was worse than the baseline three days in a row. The blue dot had the same models through the whole period. The gray dot-line shows mean and 95% confidence interval of performance generated from 100 random draws from a pool of 3k tests (daily). Since all scenarios that are compared here have the same number of tests, we can directly compare the UFS count instead of UFS recovery rate.*

Top-K approach with periodic retraining

To address the random-draw effect, we have decided to use continuous prediction values instead of binary labels (failure or pass). This way, we can rank the tests and choose tests that are more likely to fail (prediction probability for supervised learning) or more different (anomaly score of unsupervised learning models). For the supervised learning models, the default probability for binary decision-making is 0.5 and for scikit-learn's isolation forest, the threshold is 0 and negative values are all considered outliers; we can increase the threshold for the supervised and lower it for the unsupervised. Since we have an allowance in terms of the number of tests (*K* tests) we can afford to run in our ML flow, we can use these continuous scores to rank the tests, and then select the top K test candidates and only run those.

Once we fix the number of tests we run everyday, we can also simulate random-draw by using the existing random flow to compare the results between the model-selected K tests and randomly-drawn K tests. For instance, if we have run a set of 3k tests through the random flow, we randomly drawn K tests (K < 3k) multiple times and compute the summary statistics of the random draws. To compute the performance of model-selected tests, we provide the input of the 3k tests to the model and can easily compute the metrics since these 3k tests are already run and we have the labels. This comparison is shown in Fig. 7 (post-deployment, 36 days). The orange cross and the blue dot shows the performance of top K tests (K=400). The orange cross is from a scenario where we retrain the model whenever we have three consecutive *bad* days (i.e., model performance is lower than the random flow performance). The blue dot is where we never retrained the model over time. The gray dot and line indicates mean and 95% confidence interval of randomly-drawn K tests (100 times). Since every scenario in the legend has the same number of tests (K=400), it is possible to compare the absolute number of UFS (y axis, higher the better). Although models do not always perform better than the baseline, when it does (the mid section of Fig. 7), retraining the model based on our criteria did help. Considering the fact that this comparison was retrospective analysis by using the 3k tests collected daily, the top-K approach can potentially bring more benefit if we provide more tests to the models.

Conclusions

In real-world scenarios, it is often the case where one just does not have the complete freedom of algorithms or infinite amount of training resource. In hardware verification, the fact that tests are generated randomly challenge building machine learning models because we can neither guide test generation nor measure stochasticity easily. In addition, machine-learning approach is only useful when the design is mature and the majority of the tests that are run are pass but engineers are looking for failures, meaning the severe class imbalance of the training data. Finally, we cannot rely on single metric because our complementary flow competes against the existing workflow.

To address these issues, I have built a prototype that provide test candidates and filters out failure-prone tests instead of trying to guide the testbench itself, used both supervised and unsupervised models to address the problem as classification and outlier detection at the same time, customized the process of how to select the best model by looking at multiple metrics, and explore the idea of using continuous predictions instead of the binary to filter fewer but better candidates. I have also conducted experiments to address the details of retraining and identifying the cause of performance instability, which are often overlooked but crucial in post-deployment process. In summary, this work provides practical information when building a machine learning engineering product for hardware verification, where machine learning approaches are still relatively new.

References

[Wil05] Wile, Goss, & Roesner. 2005. Comprehensive functional verification: The complete industry cycle (Systems on silicon), Morgan Kaufmann Publishers Inc.

[Ioa12] Ioannides & Eder. 2012. Coverage-directed test generation automated by machine learning - A review. ACM Trans. Design Autom. Electr. Syst..

[Mam16] Mammo, Furia, Bertacco, Mahlke, & Khudia. 2016. BugMD: automatic mismatch diagnosis for bug triaging. In Computer-Aided Design (ICCAD), 2016 IEEE/ACM International Conference.

[Ber13] Bernardeschi, Cassano, Cimino, & Domenici. 2013. GABES: A genetic algorithm based environment for SEU testing in SRAM-FPGAs. Journal of Systems Architecture. 59-10, Part D.

[Cru13] Cruz, Martinez, Fernández, & Lozano. 2013. Automated functional coverage for a digital system based on a binary differential evolution algorithm. Computational Intelligence and 11th Brazilian Congress on Computational Intelligence (BRICS-CCI & CBIC).

[Bar08] Baras, Dorit, Fournier, & Ziv. 2008. Automatic boosting of cross-product coverage using Bayesian networks. Haifa Verification Conference 2008: Hardware and Software: Verification and Testing.

[Wag07] Wagner, Ilya, Bertacco, & Austin. 2007. Microprocessor verification via feedback-adjusted Markov models. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. 26-6.

[Fin09] Fine, Fournier, & Ziv. 2009. Using Bayesian networks and virtual coverage to hit hard-to-reach events. International Journal on Software Tools for Technology Transfer (STTT). 11-4, 291-305.

[Sud08] Sudakrishnan, Madhavan, Whitehead, & Renau. 2008. Understanding bug fix patterns in verilog. Proceedings of the 2008 international working conference on Mining software repositories. 39-42.