2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Guiding Functional Verification Regression Analysis Using Machine Learning and Big Data Methods

Eman El Mandouh, Mentor Graphics Siemens Business, (eman_mandouh@mentor.com)

Laila Maher, Moutaz Ahmed, Yasmin ElSharnoby, Cairo University, (Laila.Hassan94@eng-st.cu.edu.eg, Moutaz.Ahmed92@eng-st.cu.edu.eg, Yasmin.Din94@eng-st.cu.edu.eg)

Amr G. Wassal, Cairo University, (wassal@eng.cu.edu.eg)

***Abstract*— As the size of hardware (HW) design increases significantly, massive number of regression tests are required to validate its functional correctness with a huge amount of data generated during the design simulation for every test scenario. Debugging regression failures is a tedious and time consuming task. This paper addresses the challenge of reducing the debugging effort required to validate the changes between HW model iterations of the same design. It proposes the utilization of clustering machine learning technique to learn the design good behavior through the passing regression tests. Good tests grouped into testcases clusters such that the buggy test can be detected when it failed the assignment to any of the identified test clusters in the design regression test suite. Our framework utilizes x-means clustering techniques to identify the trace segment, design module name as well as design signals which are suspected to be the culprit of the bad behavior tests. Additionally, our work tackles two main challenges in the learning process. Firstly, signal selection step is done to decide which design signals should be included as the test features during the machine learning phase. Signal selection is based on the signal type as well as its connectivity network. Secondly, Big-Data processing technique, namely, Map-Reduce is used to overcome the challenge of processing huge trace dump resulted from design simulation. Our experimental results demonstrate the feasibility of the proposed approach to detect multiple design injected faults using mutation testing with HW designs.**

Keywords*— *Functional Verification, Regression Analysis, Machine Learning, Big Data

I. INTRODUCTION

One of the most time consuming and challengeable task in the functional verification cycle is debugging the failures and analyzing the massive amount of data that is produced from design simulation. Simulation sessions run thousands of tests for a long time to exercise the complete behavior of the design. As a result, thousands of GBs of debugging data are analyzed by the verification engineers to examine the huge full design trace history and verify repeatedly the results. A critical challenge in the manual debugging effort occurs when the execution result mismatches the reference behavior for the design under verification. The verification engineer must identify the root cause of the bug and narrow down the problem by recognizing the cycle and the critical design signals involved in the bug occurrence. Clearly, any attempt to automate bug detection with the execution trace window in which the buggy behavior is observed will be of great help to accelerate the debugging effort. Additionally, identifying the design module and the suspected signals which own the buggy behavior will be a plus added value. This work leverages the power of machine learning techniques to learn the correct design behavior from passing regression test execution traces. The learnt model is then used to differentiate the failing behavior and identifying the trace window, design module as well as the signals that are the root of the buggy behavior. We demonstrate the utilization of a distributed big-data processing techniques, namely MapReduce to overcome the challenge of processing huge traces result from the execution of the long tests to accelerate the data encoding step for ML operation. Finally, we control the number of design signals to be considered during feature selection for the ML step using the signal selection algorithm. Signal selection is based on the Cone of Influence (COI) analysis to identify which signals to observe during design simulation.

The rest of the paper is organized as follows. Section II briefly reviews the previous work of analyzing execution trace for anomaly detection. Section III formulates the problem and explains our proposed framework. Section IV

demonstrates the feasibility of our approach against a real design case and explains the use of mutation testing as our bug injection method. Finally, concluding remarks and future work directions is given in section V.

II. Related Work

Recent improvements in design verification strive to automate the error-detection process and greatly enhance engineers' ability to detect functional errors. However, the process of diagnosing the cause of these errors and fixing them remains difficult and requires significant ad-hoc manual effort. Many of the previous attempts of bug-localization from the dynamic analysis of HW design simulation are inspired by a prior work for the automatic detection of software program invariants using the dynamic analysis of program runs [1] [2] [3] [4] to list a few. The work in [5], Inferno uses the principle of least astonishment to understand what constitutes the common behavior of a system and use this to detect any anomaly in the design. It analyzes the results of the design's simulation trace and automatically extract high level diagram representing the design's transaction activity across all design interfaces. These diagrams are used for automatic generation of design checkers that can then be used to detect any future bad changes in the design behavior. The bug localization algorithm in [6] is based on statistical analysis of dynamically covered HDL code items (statements, branches and conditions) of processor designs. It presents an approach that is based on two main iterative phases: dynamic slicing and a statistical ranking of the HDL statements in the design that are suspected to be the root cause of the error. The dynamic slicing reduces the debugging analysis to all the statements that actually affect the design's faulty behavior for a given stimulus. Then, the suspicion ranking assigns a suspicion score to each statement present in the dynamic slice. Similar approach is used in [7] where information from regression suite results about failed and passed testcases and a number of statements executed by each test is used to find the highly candidate statement that may be responsible for this incorrect behavior. In this work, we introduce a clustering based machine learning approach to learn the correct behavior of the design under test using the entire design regression passing testcases. Any new failing test case that fails the assignment to any of the learned test clusters is recognized as a buggy test. The bug localization module will identify the buggy trace segment time associated with the design module and the signals related to the detected bug. Our work leverages the use of Map-Reduce method is to accelerate the processing of the huge resulted execution trace from the design simulation during trace encoding and feature extraction for machine learning operation.

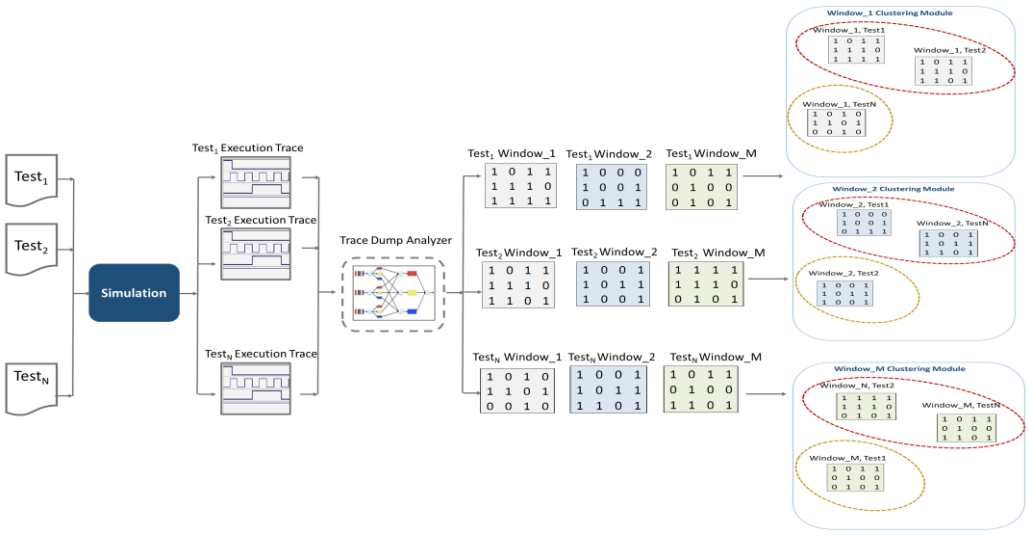

Figure 1. ML Based Regression Testing Bug Detection Proposed Framework

III. Proposed Framework

Figure 1, describes the major building blocks of the proposed framework. It starts with the RTL regression simulation step. The execution traces are stored in a comma separated value format for the trace dump analyzer. The trace dump analyzer processes the trace file for data encoding and feature extraction. The trace dump analyzer

is based on Map-Reduce method to speed up the processing of huge data resulted from the design simulation. Clustering Machine Learning is used to train cluster modules for the design good behavior using passing regression tests. Once the model is developed any new test is checked against the recognized clusters. If the test failed to be assigned to any of the recognized clusters it is identified as buggy test. Nearest-Neighbor Classifier is then used to assign the buggy trace window to the class of its closest neighbor in the feature space. The list of suspected buggy design signals are detected by differing the buggy trace window with the nearest neighbor signal features. Additionally, the design module name and the trace window number for the observed buggy behavior are also reported to the verification team. The following subsections explain in more details the main steps in our proposed solution for bug detection module as described in Figure 1.

A. Data Pre-Processing and Feature Extraction for Machine Leaning Operation

Clustering is un-supervised machine learning method. Unsupervised learning is a task of inferring hidden structure from "unlabeled" data. In cluster learning, each cluster is identified by its center (centroid) as well as its shape. The main goal of clustering is to assign similar input data points to groups such that all the data points within the same cluster are more similar to each other than those in other clusters [8].

Equation (1) demonstrates a typical data set for any clustering algorithm. Typically a vector of N input data-set X is used to extract feature vectors $h_0(x_i)$, $h_1(x_i)$,….. $h_D(x_i)$ for each input x_i, where D is the number of input features. The extracted features are then fed to train the clustering model which is going to predict for any future input test case (x_i) a cluster label assignment (y_i).

$$X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ \vdots \\ x_N \end{bmatrix} \rightarrow H = \begin{bmatrix} h_0(x_1) & h_1(x_1) & h_2(x_1) & \dots & h_D(x_1) \\ h_0(x_2) & h_1(x_2) & h_2(x_2) & \dots & h_D(x_2) \\ h_0(x_3) & h_1(x_3) & h_2(x_3) & \dots & h_D(x_3) \\ h_0(x_4) & h_1(x_4) & h_2(x_4) & \dots & h_D(x_4) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ h_0(x_N) & h_1(x_N) & h_2(x_N) & \dots & h_d(x_N) \end{bmatrix}, Y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ \vdots \\ y_N \end{bmatrix} \dots (1)$$

In our problem formulation, the testcases trace dumps are virtually divided into "N" trace windows where every window has a fixed length of simulation cycles. A separate clustering model is built for every trace window (wi), where the inputs are the regression test case trace window dump and the features are the number of every design signal value changes to 1'b1 within this trace window [9]. To reduce the number of design signals that are incorporated in the feature extraction step, our proposed framework conducts a light RTL static analysis to suppress design elements such as memories, counters and equivalent signals from being included in the feature selection step. It focuses on the selection of design control signals, FSM state signals and internal registers with large number of design registers in their cone of influence. Accordingly, there is no need to record any design signals except the ones used for trace windows encoding and feature extraction steps. Our main goal is to pick up a subset of design signals with a good probability to propagate the functional error during simulation.

RTL design elaboration at the beginning of the simulation [10], is used to extract some useful static attributes about the design under verification, such as signal types, design elements such as counters, memories, clocks or reset signals. Cone of influence analysis is done to identify which design signals are influencing the logic value changes for another design signal. Our approach builds signal connectivity graph by identifying recursively the cone of influence of each design signal. A standard network analysis algorithm is run on the graph and scores signals based on their influence score. Our algorithm uses a DFS (Depth First Search) tree rooted at each design signal and scores its influence by calculating the number of child nodes. Signals with influence score greater than a threshold value are picked up for feature extraction step.

To accelerate feature extraction step, feature extraction is done for all clustering modules all at once using Map-Reduce [11]. The MapReduce is a framework that was first introduced by Google. It parallelizes problems that require large datasets processing using a large number of computing nodes. There are many state-of-the-art frameworks to automatically schedule parallel map and reduce functions on distributed system to manipulate big input date, our approach uses Apache Hadoop [12]. Figure 2, demonstrates how MapReduce is used for feature

extraction step. The MapReduce consists of three main steps: Mapper, Shuffle Step and Reducer. MapReduce operation starts by dividing large trace dump file into blocks of 128Mbs, every data block is assigned a computing node (Mapper). The list of design signals whose values are logged in the simulation trace dump is stored as a header file that is shared among all processing nodes (Mappers). The trace chunk at each mapper is then divided by applying a sliding window of width (wi) on the trace sequences, as shown in Figure 3.

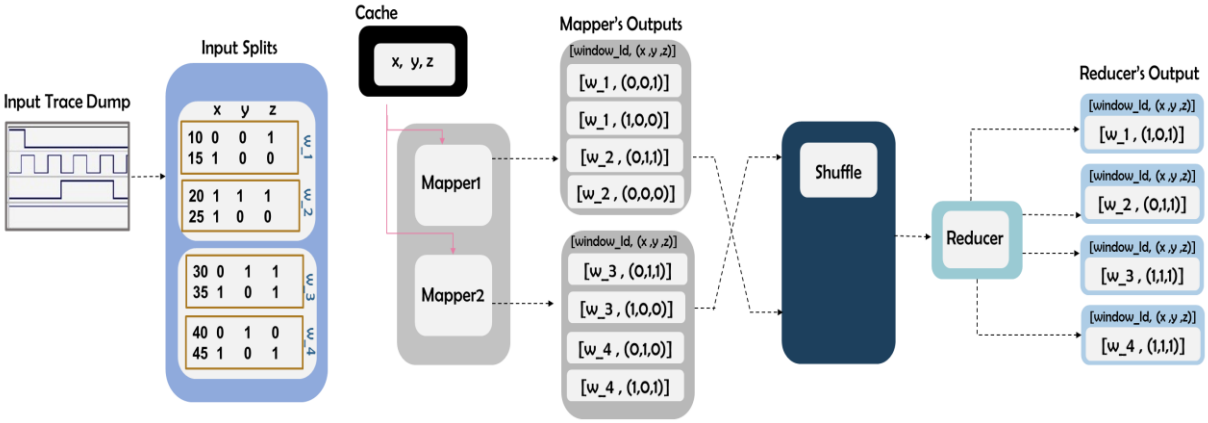

Figure 2: MapReduce For Feature Extraction Step

The map function processes each individual line in the input trace dump window (wi). The design signal values are checked against their previous values. For each signal, if the signal value does not change from its previous value, it will not be considered for feature extraction step. Feature extraction counts the number of times the signal value changes to 1'b1 within every trace window wi. The mappers will read the trace dump chunks in parallel and emit a key-value pair, where the key is the window_id (wi) and the value is the count of design signals values change to 1'b1 within that window. The map step is followed by shuffle step that Hadoop does automatically to sort and consolidate intermediate data from all mappers and before reduce tasks start. During this step every (key, value) pair is assigned a computing node such that all the occurrence of the signal changes for window (wi) lands on the same machine. The final step is the Reduce step, where the aggregation sum is done to collect all transitions to 1'b1 counts for every design signal within same trace window wi .

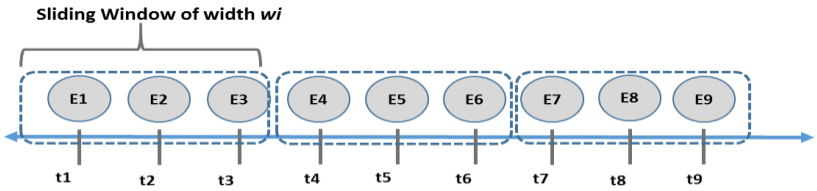

Figure 3: Dividing execution trace into trace windows: t_i is the simulation cycle time, E_i is the list of design signal value changes at t_i

B. Clustering Machine Learning

The main functionality of the bug detection module is to group similar trace windows across all regression tests into clusters such that buggy tests can be identified as outliers which fail to be assigned to any of the identified clusters. In our problem formulation, a separate clustering model is built for every trace window, wi. Clustering machine learning methods aim to group a collection of patterns into clusters based on similarity metric. Similarity is a measure that reflects the strength of the relationship between two data items, it represents how similar two data patterns are. This similarity measure in most applications is based on distance functions such as Euclidean distance. Generally, there are a lot of clustering algorithms present in the machine learning literature [8], our framework utilizes an extension of the traditional k-means clustering method. The objective of k-means is to assign input data points to the appropriate cluster that minimizes the accumulated distance from each element in the cluster to its centroid. K-means algorithm starts by random initialization for the cluster centroids. It then calculates the distance between every input data point and the k-clusters centroids. The input data is assigned to the corresponding cluster

with minimum distance from its centroid. A second iteration starts by updating the position of cluster centroids using the data points that have been assigned to them in the previous iteration. The new cluster centers are obtained by calculating the mean of the cluster data points. The algorithm iterates the assignment of data points to the new clusters centers followed by cluster centers update until it converges. One of the main challenges in k-means is to decide on the best cluster count (k). Our proposed bug detection framework utilizes X-Means clustering algorithm [13], which is an extended K-Means that tries to automatically determine the number of clusters based on statistical score, Bayesian Information Criterion (BIC). It starts with only one cluster, goes into action after each run of K-Means, making local decisions about which subset of the current centroids should split themselves in order to better fit the data. The splitting decision is done by computing the (BIC) as explained in [13]. Euclidean distance, equation 1, is the similarity metric that is used within our K-means clustering algorithm. Where X_q, X_k are the feature vectors for two input data points.

$$\sum_{i=1}^{D} X_q[i].X_k[i] \ldots (1)$$

```
Localize Bug Algorithm:
//Input: Training Data, Test Date, Start-End Time Intervals within Trace Dump
//Output: Error Trace Windows, Signals List
1:  Procedure Localize_Bug (Training_data, Test_data,w):
2:        err_win = []
3:        err_signals = []
4:        foreach i in w      // w: Trace Windows Intervals Array
5:              Training_Data_Windows[w_i] = Split Data (Training_data, w_i)
6:              Test_Data_Windows[w_i] = Split Data (Test_data, w_i)
                //Train Cluster Model using Pass Regression Tests
7:              Cluster_Model[w_i] = X-Means_Clustering
(Training_Data_Windows[w_i], Max_Cluster_Count)
                //Get Cluster Assignment for Test Data Using Trained Model
8:              Test_Data_Clusters[w_i] = Cluster_Model (Test_Data_Windows[w_i])
9:              Foreach t in Test_Data_Clusters[w_i]
10:                   If Test_Data_Clusters[w_i][t].Cluster_ID == -1
11:                         err_win = err_win.append(i)
12:                   endif
13:             endfor
//Start Scanning Error Prone Time Windows
14:       foreach i in err_win.append
                //Check Closest Cluster to Buggy Window
15:             Nearest_Good_Window[i] = 1NN_Classifier
(Training_Data_Windows [w_i], Test_Data_Windows [w_i])
16:             err_signals[i] = Diff (Test_Data_Windows [w_i],
Nearest_Good_Window[i])
17:       endfor
18:       endfor
19:       return err_win, err_signals
20: end procedure
```

Figure 4: Bug Localization Algorithm

Figure 4, explains the bug localization algorithm, it starts by splitting the training data and testing datasets into multiple trace windows. For all trace windows (wi) across the training dataset, X-Means Clustering model is used to group similar trace windows into the same cluster. Once the clustering models are generated. They can be used to predict for any new test whether its trace windows belong to any of the identified clusters or not. If the trace window fails the assignment, it is identified as buggy trace window. One Nearest-Neighbor Classifier [14] is used to assign the buggy trace window to the class of its closest neighbor in the feature space. The list of suspected buggy design signals are detected by differing the buggy trace window with the nearest neighbor signal features. Our method reports the design module name, the signal's list and the trace window number for the observed buggy behavior

IV. EXPERIMENTAL RESULTS

Our experimental result utilizes Ethernet MAC IP Core downloaded from [15]. The Ethernet IP Core is a MAC (Media Access Controller). It connects to the Ethernet PHY chip on one side and to the Wishbone SoC bus on the other. The training data constitutes of 1,500 testcases among the different test classes of the design, namely, Register

Access, Register Reset, Transmitter Packet, Transmitter CRC and Transmitter-Receiver. These 1,500 tests were generated by feeding different seeds to the Ethernet constraint random System Verilog testbench environment. The test lengths ranged from about 1,167 to 2 million cycles. The size of trace dumps varying from 40MBs up-to 5.5 GBs, Figure 5.

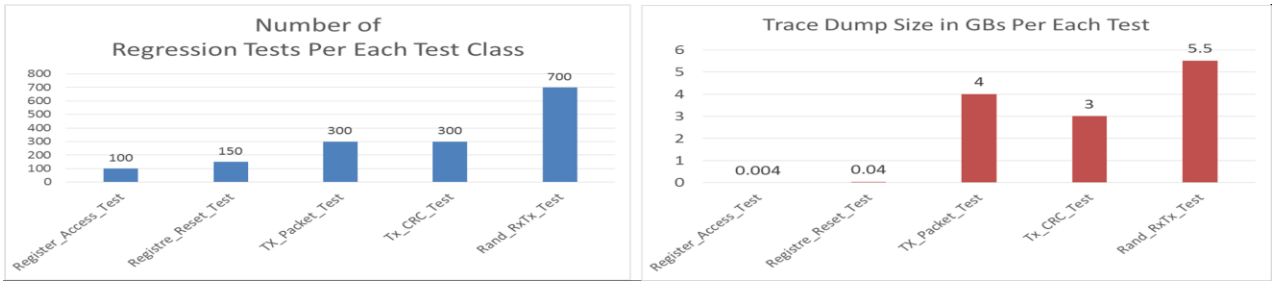

Figure 5: No of Regression Tests in Ethernet across Different Testing Classes and the Size of Each Test case Trace Dump in GBs

Recall that our workflow creates a clustering model for every trace window (*wi*). Our algorithm uses trace windows of size 100 cycles. Table 1, lists the number of trace windows extracted from every regression test. The number of windows depends on trace cycle numbers divided by the width of the sliding window (100 cycles). The clustering step in our approach utilizes X-means Clustering model to group similar trace windows of (*wi*) into the same cluster. X-means automatically determines the number of clusters based on statistical score, Bayesian Information Criterion (BIC). Maximum number limit of clusters for X-means operation in our experimental work is set to = 20.

Table 1: Training Tests Chracteristics

Test Class Name	Number of Regression Tests Per Each Test Class	Test Length in Cycles	No of Windows
Ethmac_Register_Access_Test	100	1,167	12
Ethmac_Registre_Reset_Test	150	13,071	131
Ethmac_TX_Packet_Test	300	1,509,759	15098
Ethmac_Tx_CRC_Test	300	1,021,719	10217
Ethmac_Rand_RxTx_Test	700	2,000,000	20000

Clustering machine learning starts with the feature extraction step. Our model utilizes the number of times the design signal value changes to 1'b1 within the simulation trace window as the main feature metric. In order to reduce the number of design signals included in the feature extraction step and hence reduce the clustering model complexity, our framework utilizes COI analysis as well as RTL static analysis as the main methods to select important design signals that have a large number of signals in their cone of influence logic with a good probability to propagate the functional error during simulation. COI score for a given design signal is defined as the number of signals in its COI logic. Signals with influence score greater than a threshold value are picked up for feature extraction step. This threshold value is calculated by examining the distribution of design signal influence scores and selecting a midway threshold value to prune the connectivity graph in the first iteration. This process is iterated until reaching a threshold value that achieves a good reduction ratio and sustains good coverage for design signals. Figure 6, demonstrates the COI score distribution for the Ethmac design signals (2342 signals). In this experiment, we pick up a threshold value of "10" which reduces the number of design signals included in the feature extraction, but guarantee good coverage for design value changes during the simulation run.

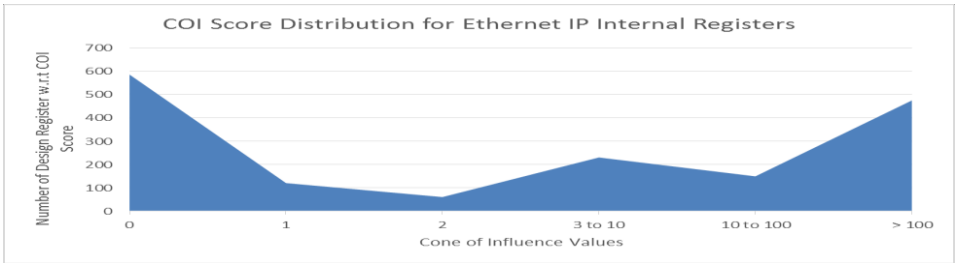

Figure 6: COI score distribution for Ethmac Design Signals

Recall that the Bug Localization Module utilizes the distributed data processing method, namely Map-Reduce, to accelerate the design trace processing and feature extraction for ML modules. During our experiment we built Hadoop cluster of four nodes (3.4GHz Intel Core i7, Quad Core, and 16GBs Memory). Figure 7, plots the processing time in seconds of the trace dump encoding and feature extraction steps for the five test classes of the Ethernet IP with and without the use of Map-Reduce. Figure 7, demonstrates that the Map-Reduce accelerates the trace dump encoding as well as feature extraction steps by a factor of 2.7x faster than the sequential data processing techniques.

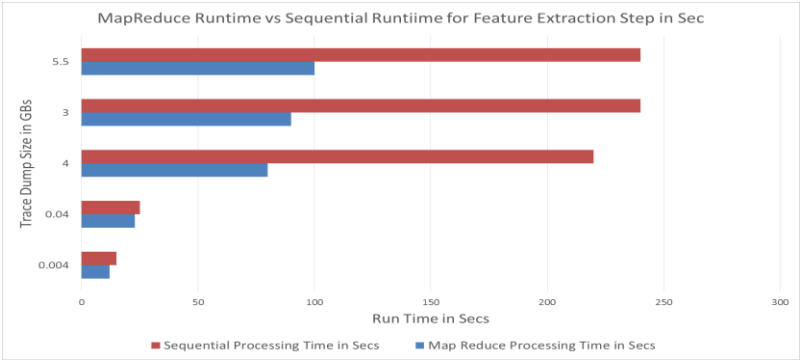

Figure 7: MapReduce Speedup for Feature Extraction Step

The testing dataset has been generated by injecting bugs to the design RTL using mutation-based-testing technique. Mutation-based testing originated in the early 1970s in software research to guide the software testing towards the most effective test sets possible [16]. A "mutation" is an artificial modification in the tested program, induced by a fault operator. It changes the behavior of the tested program. Our mutation generator creates a set of faulty versions of the original description by injecting one fault per version. These faults are small and syntactically correct modifications of the original instructions. The set of the used mutation operators, the mutated design modules, signals and the number of bugs injected are listed in Table 2a. The logical and Unary operators mutations are done based on the work published in [17] and summed at Table 2b.

Table 2a.) Mutated Modules, Signals, Injected Bugs for Testing Data-Set, 2b.) Mutation Operators Replacement Scheme

Mutated Module	Mutated Signals	No of Injected Bugs	Mutation Operators Used
eth_maccontrol	MuxedAbort	11	Logical operator replacement
eth_macstatus	LatchedCrcError, InvalidSymbol,	27	constant replacement Logical opreator replacement
eth_rxaddrcheck	MulticastOK	2	logical operator replacement unary operator insertion
eth_transmitcont	BlockTxDone	6	constant replacement
eth_wishbone	Flop, BDWrite, TxBDReady	19	Logical operator replacement Relational operator replacement
ethmac	TxPauseRq_sync1 TxPauseRq, RxAbort_latch, Collision_Tx2, TPauseRq	24	unary operator insertion unary operator removal logical operator replacement
wishbone	TxEn_needed	9	constant replacement

Name	Operator/Expression	Mutation
Logical	{! , && , ||}	Replaced to { Remove Negation, ||, &&}
Bitwise	{~, &, | , ^}	Replaced with { Remove Negation ,|,&,&}
Arithematic	{+ , - , * , / }	Replaced with { - , + , / , *}
Equality	{<, <=, ==, >= , >, !=}	Replaced with { >, >=, !=, <=, < , ==}

To judge the effectiveness of the proposed framework to help in analyzing the regression test failures. A group of faulty regression tests across all the test classes have been introduced to the clustering ML framework to identify the buggy trace window, the name of buggy signals and modules. Model accuracy reflects the ability of the model to make correct predictions. The accuracy of the created model is judged by counting the ratio of tests for which the clustering model managed to detect the exact injected bugs (Success) or to detect a superset of design signals that contains the buggy signals and other un-impacted signals (Partial Success) vs the number of times the clustering module failed to detect the injected bugs in the faulty regression tests. Additionally, we calculate the bug detection latency which is the difference between the cycles where the bug has been injected versus the cycle in which it has been detected. Figure 8, plots the model accuracy for all test classes, in indicates that on average our proposed framework demonstrates an average of 81% full success, 13% partial success and 6% failure ratio of the model

ability to detect injected bugs in the design regression tests. Additionally, figure 8, indicates that on average 83% the bugs are detected in the same clock cycle of the bug injection cycle.

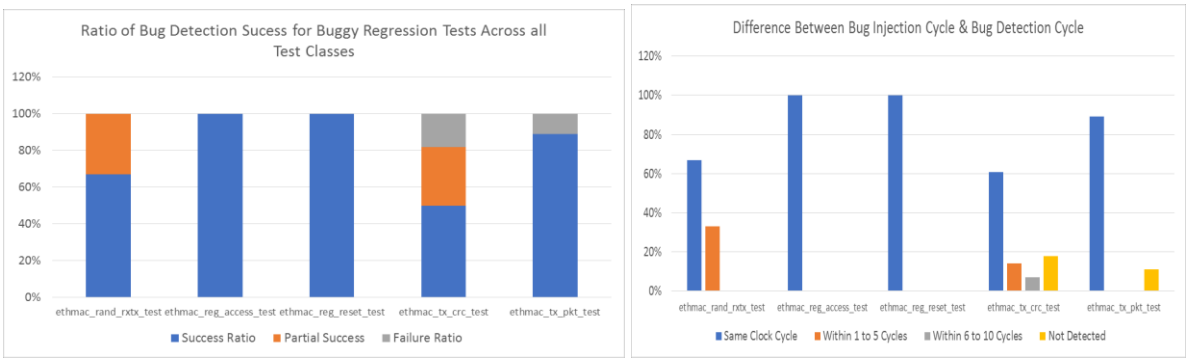

Figure 8: Bug Detection Success and Bug detection latency across all Test Classes

V. CONCLUSION

This paper proposes a framework that facilitates functional verification regression failure analysis. It utilizes clustering machine learning to learn the design good behavior through the passing regression tests. Good tests grouped into clusters such that any buggy test can be detected when it failed the assignment to any of the identified test clusters of the design regression test suite. The proposed bug detection solution can identify the buggy trace segment time associated with the design module and the buggy signals list. Map-Reduce technique is used to accelerate the processing of the largely generated simulation trace dump during the feature extraction step. We demonstrate an average of 2.7x speed up using Map-Reduce versus traditional sequential data processing methods. Additionally, our results indicate that the bug-detection module manages to detect the injected bug with up to 81% accuracy.

REFERENCES

[1] S. Hangal and M. S. Lam, "Tracking Down Software Bugs Using Automatic Anomaly Detection," In the Proceedings of the 24rd International Conference on Software Engineering, ICSE, 2002, pp. 291-301.

[2] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, M. I. Jordan, "Scalable statistical bug isolation", In the Proceedings of Programming Language Design and Implementatio n, PLDI 2005, pp.15-26.

[3] W. E. Wong , Y. Qi, "BP neural network-based effective fault localization", In the Proceedings of International Journal of Software Engineering and Knowledge", 2009, pp. 573-597.

[4] W. E. Wong, V. Debroy, B. Choi, "A family of code coverage based heuristics for effective fault localization", In the Proceedings of Journal of Systems , 2010, pp.188-208.

[5] B. Isaksen and V. Bertacco. 2006. Verification through the Principle of Least Astonishment. In the Proceedings of the International Conference on Computer-Aided Design (ICCAD). pp 860–867

[6] A. Tsepurov, M. Jenihhin, J. Raik, G. Bartsch, J. Escobar, H. Wuttke ," Localization of Bugs in Processor Designs Using zamiaCAD Framework", In the Proceedings of 13th International Workshop on Microprocessor Test and Verification, 2012, pp 41-47

[7] K. Salah, "New Trends in RTL Verification: Bug Localization, Scan-Chain-Based Methodology, GA-Based Test Generation",In Design Verification Conference, Dvcon 2015.

[8] E. Fox, C. Guestrin, "Machine Learning : Clustering and Retrieval", https://www.coursera.org/learn/ml-clustering-and-retrieval/home, 2016

[9] A. DeOrio, Q. Li, M. Burgess, V. Bertacco, "Machine Learning based Anomaly Detection for Postsilicon Bug Diagnosis", In the Proceedings of Design Automation and Test in Europe, DATE, 2013.

[10] Questa Advanced Simulator, https://www.mentor.com/products/fv/questa/

[11] J. Dean and S. Ghemawat "MapReduce: Simplified Data Processing on Large Clusters",OSDI'04: ,In the proceedings of Sixth Symposium on Operating System Design and Implementation(OSDI'04), 2004

[12] T. White, "Hadoop: The Definitive Guide", 2015, 4th Edition, ISBN: 978-1-491-90163-2

[13] D. Pelleg , A. Moore "X-means: Extending K-means with Efficient Estimation of the Number of Clusters", In Proceedings of the International Conference on Machine Learning, ICML,2000.

[14] M. Narasimha,V. Susheela "Pattern Recognition, An Algorithmic Based Approach", 2011,Springer 978-0-85729-495-1, pp 48-85

[15] Opencres benchmarks "http://opencores.org".

[16] Y. Serrestou, V. Beroulle and C. Robach,"Functional Verification of RTL Designs Driven by Mutation Testing Metrics", In Proceedings of Digital System Design Architectures, Methods and Tools(DSD) , 2007, PP 222 – 227

[17] Mutation operators listing http://pitest.org/quickstart/mutators.