Journal of Electronic Testing
https://doi.org/10.1007/s10836-018-5716-y

Application of Machine Learning Techniques in Post-Silicon Debugging and Bug Localization

Eman El Mandouh[1] · Amr G. Wassal[2]

Received: 30 December 2017 / Accepted: 6 March 2018

Abstract
As the size of hardware (HW) design increases significantly, a huge amount of data is generated during the design simulation, emulation or prototyping. Debugging large HW designs becomes a tedious, time consuming and a bottleneck task within the function verification activities. This paper proposes the utilization of machine learning techniques to automate the diagnosis of design trace dump as well as helping in bug localization during post-silicon validation. Our framework starts by signal selection algorithm that identifies which signals to monitor during design execution. Signal selection depends on signal types as well as their connectivity network. The design is then executed and the trace dump is saved for offline analysis. Big-Data processing technique, namely, Map-Reduce is used to overcome the challenge of processing huge trace dump resulted from design running on FPGA prototype. K-means Clustering method is applied to group trace segments that are very similar and to identify the ones with a rare occurrence during the design execution. Additionally, we propose a bug localization framework in which X-means clustering is used to group the passing regression tests in clusters such that buggy tests can be detected when they fail to be assigned to any of the trained clusters. Our experimental results demonstrate the feasibility of the proposed approach in guiding the debugging effort using a group of industrial HW designs and its ability to detect multiple design injected defects using mutation-based-testing method.

Keywords Post-silicon validation · Big-data · HW debugging and machine learning

1 Introduction

As HW designs increase in size and complexity, effective techniques for post-silicon validation such as FPGA prototyping are required for better evaluation of the design functional correctness [1]. Prototyping platforms run tests in a fraction of time compared to traditional simulation based verification. However, it is a challenging task due to the combined effects of debugging complexity and limited observability [2]. One of the most difficult and time consuming tasks in post-silicon validation is the debugging and the analysis of the massive amount of data that is produced from executing SoCs as FPGA prototypes. Today, post-silicon validation is an ad-hoc process and usually includes a lot of manual effort. During post-silicon debugging the verification engineer may be interested in one of the following scenarios:

- Design debugging, when the execution result mismatches the reference behavior for the design under verification. The verification engineer must identify the root cause of the defect and narrow down the problem by recognizing the cycle and the critical design signals involved in the defect occurrence.
- Diagnostics of the huge trace dump, to identify which execution trace segments own rarely occurring behavior and hence may be of great interest to his inspection. And which segments are redundant and very similar to each other and hence reflecting steady design behavior.

We propose a framework to tackle these two challenges in post-silicon validation. Firstly, our work introduces a "Bug Localizer Module" to automatically identify the suspected buggy trace window associated with the design module and signals that may be the root cause of the observed design's bad behavior. Secondly, we introduce the "Trace Debugging Module". It helps identify the trace chunks that are similar

Responsible Editor: V. D. Agrawal

✉ Eman El Mandouh
eman_mandouh@mentor.com

[1] Mentor Graphics, Siemens Business, IC Verification Solutions, Cairo, Egypt

[2] Computer Engineering Department, Cairo University, Giza, Egypt

Published online: 15 March 2018

to each other and the unique ones that inherit rare design behavior, hence are good candidates for a closer look during the debugging cycle. Our proposed solutions depend heavily on machine learning methods as the core techniques for trace diagnostics as well as for the bug localization framework.

Figure 1, describes the major blocks in our proposed framework. It starts with the static analysis of the RTL design to select a group of signals to observe during design execution. The design is then synthesized on the target FPGA prototype system. The execution trace is stored in VCD (Value-Change-Dump) format for off-line trace analysis step. 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 execution.

Trace Debugging Module starts with a list of trace windows encoded as event sequences. Clustering-based technique is used to group similar trace chunks into the same cluster so that the debugging effort can be scheduled for each cluster accordingly. Clusters with a lot of elements indicate redundant trace chunks, and the verification engineer could pick up a few of them for inspection. Clusters with few members reflect trace chunks with unique or rarely occurring behavior so deeper debugging effort should be assigned to them. In Bug Localizer Module a clustering model is trained about the design good behavior using passing regression tests. Once the model is trained any new test is checked against the trained clusters. If the test failed to be assigned to any of the trained clusters it is identified as buggy test. The Bug Localizer Module reports design module name, the signal's list and the trace window number for the observed buggy behavior.

The main contributions of this paper are: Firstly, it leverages the power of machine learning techniques in both trace diagnosis and bug localization in post-silicon validation cycle. Next, it demonstrates the utilization of a distributed big-data processing technique, namely MapReduce to overcome the challenge of processing a large number of traces resulting from the design execution during data encoding and feature extraction steps. Finally, it presents a signal selection algorithm that is based on the Cone of Influence analysis to recommend which design signals are profitable to be included in the machine learning feature selection step, hence to be observed during post-silicon debugging.

The rest of the paper is organized as follows. Section 2 briefly reviews the previous work on analyzing execution trace for anomaly detection and the previous work in post silicon debugging. Section 3 gives a brief background about clustering in machine learning. Section 4 formulates the problem and explains our proposed framework. Section 5 demonstrates the feasibility of our approach against real design cases and explains the use of mutation testing as our defect injection method. Finally, concluding remarks and future work directions are given in section 6.

2 Related Work

This section briefly sums up the related work in automating bug localization as well as debugging in post-silicon validation.

2.1 Bug Localization in Post-Silicon Validation

Bug localization involves identifying a bug trace (a sequence of inputs that activates and detects the bug) and a hardware design block where the bug is located [3]. There are many previous attempts of bug-localization in post-silicon validation. The work in [4] presents IFRA, which is a technique for localizing electrical bugs in processors. Special on-chip recorders collect information during normal operation of a

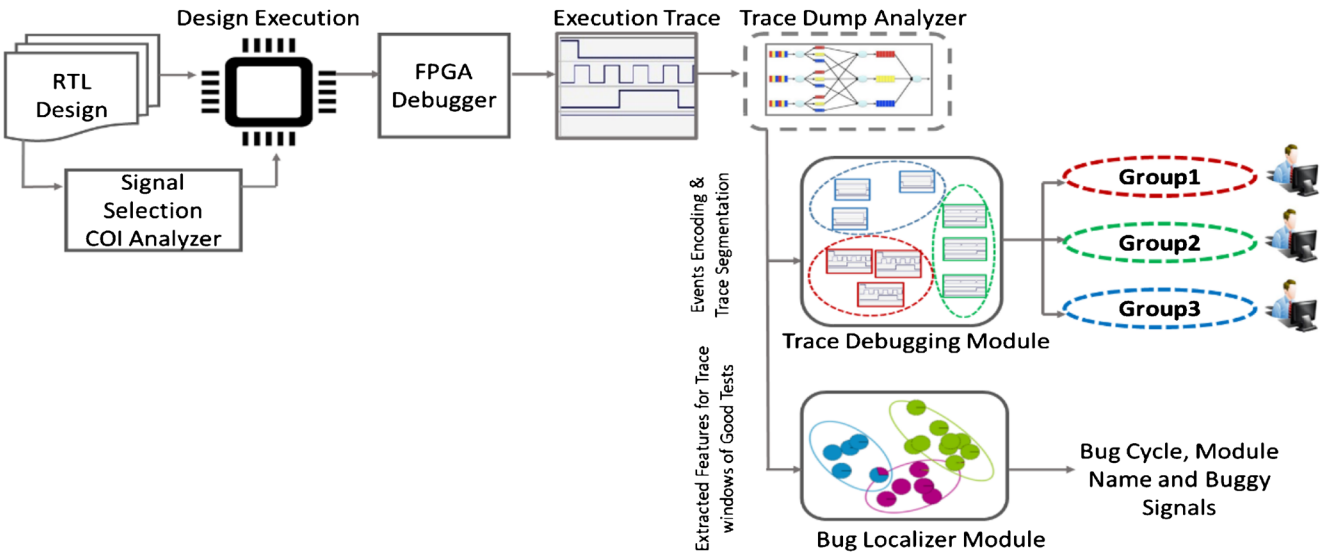

Fig. 1 Proposed machine learning based trace dump debugger & bug localizer framework

processor in the post-silicon validation setup about flows of instructions through the processor and what the instructions did as they passed through various design blocks. Upon system failure, the recorded information is scanned out and analyzed off-line using special self-consistency-based analysis techniques to localize hardware bugs. In [5] a BPS, Bug Positioning System, which is a lightweight hardware logs a compact encoding of observed signal activity over multiple executions of the same test. It then leverages a novel post-analysis algorithm, which uses the logged activity to diagnose the bug, identifying the approximate manifestation time and critical design signals. E-QED, is introduced in [3], E-QED signature blocks are used to capture and compress the logic values of selected signals. The collected signatures are then analyzed by a Bounded Model Checking (BMC) formal tool to first identify which design block produced the error(s), and then to find the FFs in that block that could have captured the error(s). The work introduced in [6] starts by converting error symptoms into assertions. It then focuses on the design logic that can be impacted by the detected bug, adjusting design input constraints and using model checking to find the bug root cause (Counterexample). In [7] the post-silicon trace analysis result is compared with a reference model. If a bug is detected SAT-based analysis is used for bug detection. This is done by enhancing the circuit with the correct model by adding multiplexers at design components out ports. The circuit behavior is complying with its implementation if the correct model is inactive, else the design component outputs are replaced with the corrected values and the diagnostic trace is generated. The work in [8] utilizes an anomaly detection method to locate the time of a bug's occurrence, and the critical signals involved during post-silicon validation. Our approach differs from the work of [8] in many aspects. Firstly, we introduce a signal selection algorithm that control which signals to log during design execution on the FPGA Prototype. Our signal selection algorithm depends on the design static analysis to identify COI of the selected signals. Secondly the work in [8] learns the good behavior of the design by multiple executions of the same test when some of these trials may pass while others fail. Our approach learns the correct design behavior by training the machine learning algorithm across the entire design regression testcases with its different test classes. Our approach utilizes X-means clustering method to automatically determine the number of clusters and hence avoid this limitation with K-means clustering technique. Unlike the work in [8] our proposed framework does not separate time localization and signal localization, which add more processing overhead to the machine learning step. Finally, our work introduces for the first time the utilization of Map-Reduce method to accelerate the processing of the huge resulted execution trace from the design prototyping. Additionally, none of the previous work tackles the challenge of debugging huge trace dump and what should be the starting point to look at. Our work introduces for the first time automatic framework for the analysis of huge trace dump that utilizes machine learning methods to guide further debugging effort. It points to the trace parts which are unique for closer inspections or redundant trace chunks where sampled inspection can be used to look at their similar trace periods.

2.2 Post-Silicon Debugging

Recall that our framework starts by signal selection step. Our signal selection algorithm is done at the register transfer level (RTL) before design synthesis to the gate level netlist. The main motive for this is to select which design signals to incorporate in the feature extraction step during the cluster-model training, hence reduces the model complexity. 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 post-silicon validation. The proposed signal selection algorithm depends on the static analysis of the RTL to suppress design elements such as memories, counters and equivalent signals from post-silicon debugging. It focuses on the selection of design control signals, FSM state signals and internal registers. It utilizes formal cone of influence (COI) to pick up the internal signals with large number of design registers in their formal cone of influence for post silicon debugging. Although a lot of signal selection algorithms were proposed previously. They are not suitable for our framework. The main reason is that almost all of the previous signal selection algorithms are done at gate level netlist with the aim to maximize the Signal Restoration Ratio (SSR) metric. They utilize a heavy analysis techniques to study the selected signals ability to restore the values of other signals in the design. The signal restoration problem is not the main problem we tackle in this work, because our debugging framework allows ten thousands of design signals to be traced using an observation network. The observation network is a non-blocking switching network that has more inputs than outputs to funnel a subset from the chosen design signals to the trace buffer at a time. Observing another set of design signals can be done at run time without a need to re-instrument the RTL design and go through the synthesis cycle. The debugging architecture will be explained in details in the next section.

This section sheds the light on the previous signal selection algorithms to illustrate the idea and demonstrate the difference between them and the signal selection step in the proposed framework. The work in [9] [10] was the first to define the term state restoration ratio (SRR) as the ratio of the number of state values restored over the state values traced for a given time interval. They also introduced the first trace signal selection algorithm that based on maximizing the amount of restored state. Further research has produced several improved solutions for automatic signal selection all sharing the goal of

improving SRR. [11] improved signal restorability by proposing an algorithm that considered already selected signals while selecting new ones. In [12, 13], simulation is used to obtain an accurate metric for state restoration by running the circuit over a small number of cycles and measuring the corresponding restoration ratio. Simulation-based algorithms are based on the intuition that if a set of signals works well for some random input vectors, then it is also likely to provide high state reconstruction on other inputs and therefore a high restorability ratio. The authors in [14] introduce a new signal selection algorithm using a logic implication-based learning approach to intelligently select the trace signals that contain more implications that are not implied by other signals. A hybrid signal selection approach combining the advantages of both simulation and metric-based signal selection algorithm has been introduced in [15]. All the above methods focus mainly on trace signals monitoring during post-silicon execution. However, scan-based debugging is another popular post-silicon debugging practice. Scan chains are used to identify fabrication defects and uses a series of interconnect registers, flip flops or sequential elements where signal values are serially extracted. However, data can only be dumped in scan mode, during which the design stops its normal execution, thus preventing real time observability of the internal design signals. To overcome this challenge, some other approaches [16, 17] explored a profitable combination of trace and scan signals. The idea is to divide the trace buffer width into two parts. The first part stores the trace signals and the second part stores the scan signals. There is a very small set of important control signals that would be traced in every cycle. The remaining slots of the trace buffer will be filled with a portion of a large set of scan signals that would be dumped across several cycles.

Our machine-learning based bug localization framework utilizes a signal selection algorithm as part of the work flow. The proposed signal selection algorithm is different from the previously discussed algorithms in many aspects. First our algorithm does not depend on signal restoration metric as the criteria for selecting candidate signals for post-silicon debug. Our signal selection algorithm is structured based approach, which uses information about the design structure to identify profitable signals for feature extraction step during model training. It nominates a group of signals with a good coverage such that the injected functional bugs can propagate easily in the captured execution trace without a need to add all designs signals for post-silicon debugging. Accordingly, the signal selection algorithm is done on the register transfer abstraction level because signal selection at RTL level is simpler and faster than gate level netlist abstraction level [18]. The COI analysis is used to identify what are the registers that influence a large number of other design register, then a connectivity graph is built to present the relation between design variables. The signals with large number of design registers in their formal cone of influence is picked for post silicon debugging.

3 Background

3.1 Clustering in Machine Learning

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.

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).

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

There are a lot of clustering algorithms present in the machine learning literature [19], our framework depends heavily on k-means clustering methods. 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, Fig. 2, the algorithm starts by random initialization for the clusters 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

```
Algorithm: K-means
//inputs: data, K Value, initial set of centroids, maximum number of iterations to run
Inputs: data, k, initial_centroids, maxiter
Outputs: centroids, cluster_assignment
    1. centroids = initial_centroids
    2. prev_cluster_assignment = None
    3. for itr in range (maxiter):
         //make cluster assignments using nearest centroids, Cosine Similarity
    4.   cluster_assignment = assign_clusters (data, centroids)
         //compute new centroid for each of the k clusters, averaging all data points assigned
         to that cluster
    5.   centroids = reassign_centroids (data, k, cluster_assignment)
         //check for convergence if none of assignments changes stop
    6.   If prev_cluster_assignment is not None &&
           (prev_cluster_assignment = cluster_assignment):
    7.     Break
    8.   endif
    9.   Prev_cluster_assignment = clutser_assignment
    10. endfor
    11. return centroids, cluster_assignment

func assign_clusters(data, centroids):
   //Compute distances between each data point and the set of centroids
1.  distances_from_centroids = Calculate_Similarity_distances(data,centroids)
   //Compute cluster assignments for each data point:
2.    cluster_assignment= []
3.    for i in range(data.len()):
4.      cluster_assignment.append(np.argmin(distances_from_centroids[i,:]))
5.    endfor
return cluster_assignment

func reassign_centroids (data, k, cluster_assignment):
 1. new_centroids = []
 2. for i in range(k):
      // Select all data points that belong to cluster
 3.   member_data_points = data[cluster_assignment==i]
      //Compute the mean of the data points.
 4.   centroid = member_data_points.mean()
 5.   new_centroids.append(centroid)
 6. endfor
return new_centroids
```

Fig. 2 K-means clustering algorithm

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.

The two main challenges in k-means clustering are cluster centroid initialization and the selection of number of clusters, k, to start the algorithm with. We will demonstrate the used techniques to overcome these challenges in Section 4.

4 Problem Formulation and Proposed FrameworK

The following subsections explain in more details the main steps in our proposed solution for post-silicon debugging acceleration as described in Fig. 1.

4.1 Signal Selection Algorithm

The proposed framework starts by the static analysis of design under verification to identify a sub-set of design signals that are effective for debugging during design execution. Figure 3, demonstrates our Signal Selection Algorithm. Signal selection is based on the signal type as well as signal connectivity. It assigns design registers, inputs, FSM state signals and design control signals higher weights than memory elements, counters, clocks and reset signals. Additionally, it depends heavily on evaluating the Cone of Influence coverage for design internal registers. The cone of influence coverage is a terminology used to describe the activation of design logic that drives particular design signal. The COI analysis step during signal selection is done using the formal verifier in [20]. Formal cone of influence produces more accurate results than the static COI. Formal COI analysis prunes down the static cone of influence to focus only on the real logic that actually drives the design signal of interest. Formal analysis incorporates logic behavior of the design as well as design constrains to define real paths between two design variables rather than the static functional paths. For example, in Fig. 5, if the "sel = 1'b0" is a design constraint or constant value, then register "H" can never be part of COI of "B". Another example, suppose that for a design variable "X [1:0]" the design behavior never reaches the two states of "2'b10, 2'b11" then formal cone of influence will exclude these unreachable design states from the COI of any depending variable on "X".

Our signal selection algorithm starts by calculating recursively the formal cone of influence (COI) of each design signal. This is followed by the creation of COI connectivity network graph for the design under verification. Where each node represents a design signal. The edge between two nodes represents the direct influence path from the source node to the destination node. Once the graph is constructed a standard network analysis algorithm is run on the graph to calculate for each node (design signal) its influence score. The node

```
Signal Selection Algorithm:
Input : Design Signals List
Output : Selected Signal for Dumping
1:  Procedure SIGNAL_SELECTION (design_signals_list):
2:        for sig in design_signals_list
3:              signals_coi = calculate_COI (sig)
4:              G = create graph (signals_coi)
5:              for node in G:
6:                    tree = DFS tree (G, node)
7:                    if tree.children > threshold then:
8:                          selected_signals.append (node)
9:                    end if
10:             end for
11:       end for
12:       return selected_signals
13: end procedure
```

Fig. 3 Signal selection algorithm

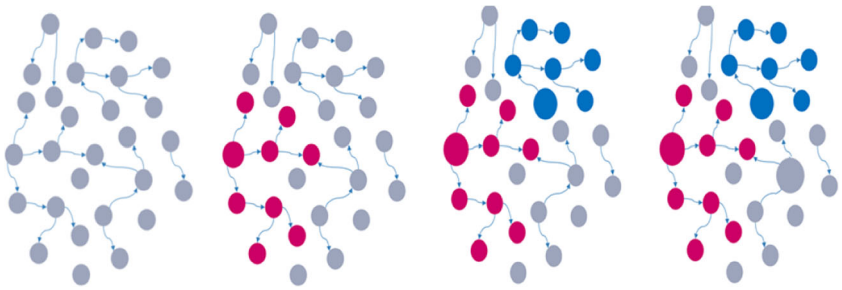

Fig. 4 COI score calculation & signal selection algorithm

influence score is defined as the number of connected signals to this node tree. The algorithm iterates over the network graph nodes and uses a DFS (Depth First Search) tree rooted at every node in the graph to count the node children as its influence score. The influence score reflects the connectivity of the design signal with other design signals throughout the design paths. A threshold value is defined such that if the node influence score is less than it the design signal is not added to the selected signals list. Else the signal is selected and the node corresponding to it as well as its connected nodes are marked as visited.

Figure 4, explains the steps of signal selection algorithm, the large sized nodes are the ones that are selected by the signal selection algorithm when the influence score threshold is set to "4". Free Python Library for Network-Graphs Generation and Data Manipulation is used within our proposed framework [21].

Figure 5, represents a simple example for a network connection graph of the circuit in the same figure. Each design variable in the circuit is represented by a node in the graph. Variable "H" is excluded from formal COI analysis. The node influence score of "C" is three, since it is connected to nodes (A, B, and G). While "E" has an influence score of four as it is connected to "C, A, B and G". If the threshold value is defined as 40%, i.e. 40% of the network graph nodes, then the algorithm will pick up all the nodes with children count equal of greater than 2.8 nodes, so it will return {C, E} as the selected signals because all other nodes have an influence score less than 2.8.

4.2 FPGA Debugging Architecture

Our work utilizes advanced post-silicon debugging framework [22], Fig. 6. It starts with the RTL design compilation and elaboration. Signal selection is done at this step where the proposed signal selection algorithm is used to nominate a group of design signals to observe during the post-silicon debugging. The original RTL is then instrumented to

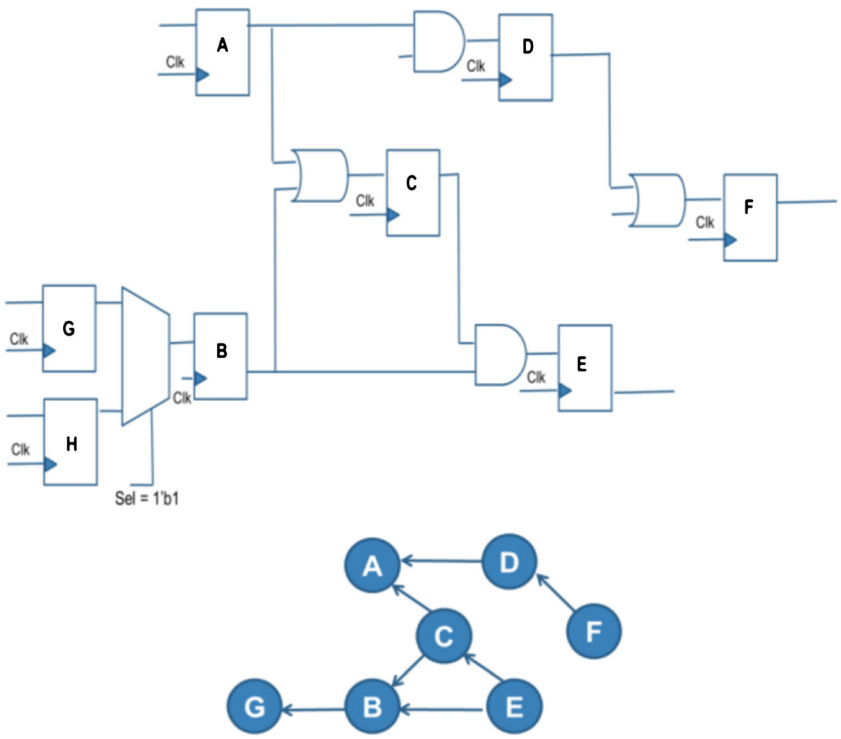

Fig. 5 Example circuit with signal connectivity network

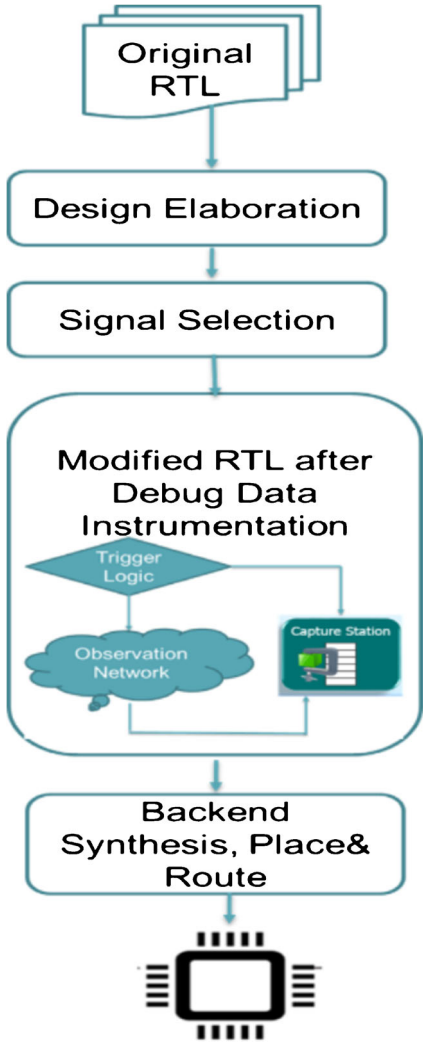

Fig. 6 Post-silicon debugging framework

externalize the selected signals using probes and add the debugging logic IPs. The debugging logic consists of three major components, namely the observation network, trigger logic and trace buffer configuration. The observation network is interconnected or switching network that allow the selection of a subgroup of signals from a larger set to be connected to the trace buffer at a time. It is very useful to maximize the number of signals captured during the design execution because users can switch between a group of design signals to watch at a time without a need to re-instrument the RTL then to go through re-synthesis cycle. The instrumented RTL which consists of original RTL, trace signal probes as well as the debugging modules are passed to the backend synthesis/place and route to the FPGA Platform. The trigger logic is the user specified condition to start store the trace logic into the trace buffer/Capturing stations. Once the capturing condition is achieved during the run time the design trace is uploaded from the trace buffer to the host machine using JTAG in VCD format.

4.3 Bug Localization Module

The Bug Localization Module starts by running a group of design regression tests (Test1, Test2..). The design execution traces for every test are stored in VCD (Value Change Dump) format. The VCD files are converted to flatten comma separated value format "csv" files that are appropriate for independent data processing during feature extraction using MapReduce. The trace dump is virtually divided into "N" trace windows where every window has a fixed length of design execution cycles. Figure 7, gives a simple example for nine cycles trace dump for design signals "a, b, c, d" across three testcases "Test1, Test2 and Test3". Suppose that the sliding window length is set to three cycles, accordingly the trace dump will be divided into three windows (windows_1, window_2 and window_3).

The main functionality of the bug localization 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, Fig. 8. In our problem formulation, a separate clustering model is trained for every trace window ($_i$), where the inputs are the ($_i$) parts of the trace dump across the entire regression tests. Feature extraction is done by counting the number of signal value changes within (w_i) for every input regression test (Fig. 9).

Figure 10, demonstrates how features are extracted for trace window_1 across the three testcases (Test1, Test2, Test3) in the example of Fig. 7. The feature vector for window_1 counts the number of times every design signal value changes to "1'b1" or "1'b0" within window_1. For example, in Test1, signal "a" has "1'b1" three times within the trace window_1. While signal "b" has the value "1'b1" twice within window_1.

To accelerate feature extraction step, feature selection is done for all clustering modules all at once using MapReduce [23]. 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

		Test 1				Test 2				Test 3			
		a	b	c	d	a	b	c	d	a	b	c	d
Window_1	T1	1	0	1	1	1	0	1	1	1	0	1	0
	T2	1	1	1	0	1	1	1	0	1	1	0	1
	T3	1	1	1	1	1	1	0	1	0	0	1	0
Window_2	T4	1	0	0	0	1	0	0	1	1	0	0	1
	T5	1	0	0	1	1	0	1	1	1	0	1	1
	T6	0	1	1	1	1	0	0	1	1	1	0	1
Window_3	T7	1	0	1	1	1	1	1	1	1	0	1	1
	T8	0	1	0	0	1	1	1	0	0	1	0	0
	T9	0	1	0	1	0	1	0	1	1	1	0	1

Fig. 7 Trace Dump Sample for Three Tests: Test1, Test2 & Test3

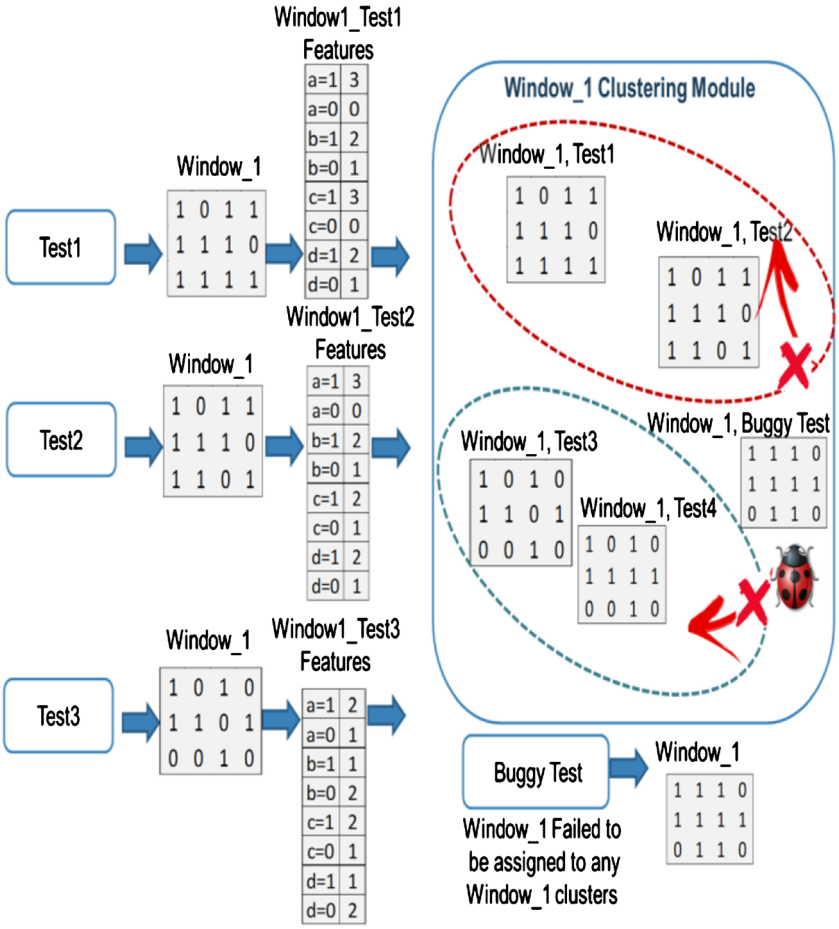

Fig. 8 Illustrative example for bug localization algorithm

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 [24].

Figure 9, demonstrates how MapReduce is used for feature extraction step, using "Test1" trace dump example in Fig. 7. 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 mappers read the trace dump blocks in parallel and emit key-value pairs. Where the key is the window_id (w_i) and the value is the count of the signal value changes to 1'b1 and 1'b0 within that window. For example, in Fig. 9, every mapper is assigned a trace chunk of

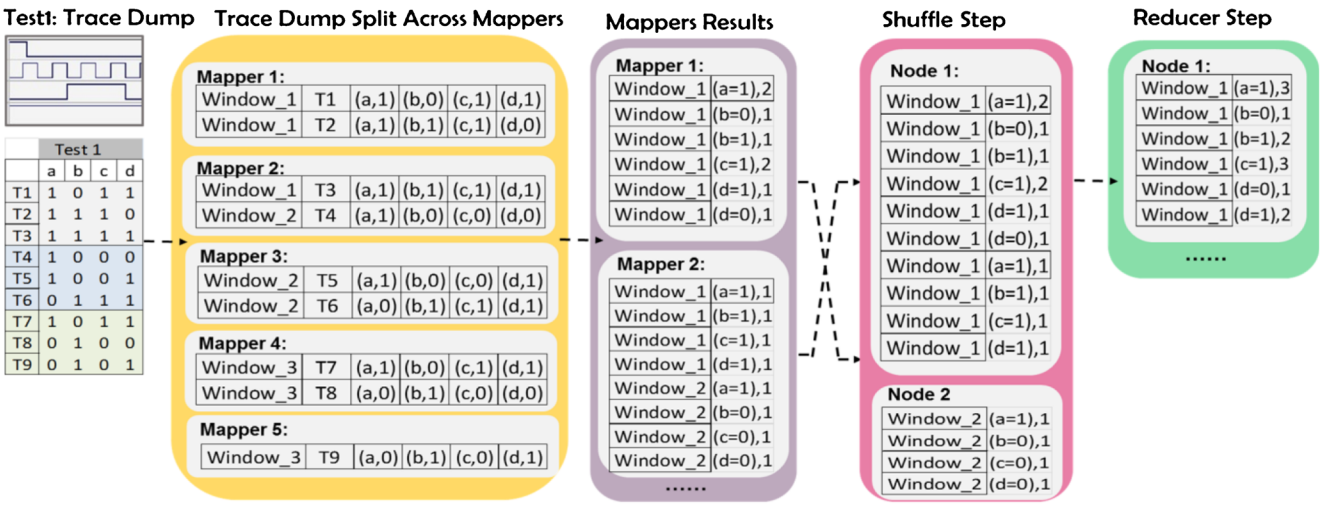

Fig. 9 Bug localization feature extraction steps using map-reduce

Window_1, Test1

a=1	a=0	b=1	b=0	c=1	c=0	d=1	d=0
3	0	2	1	3	0	2	1

Window_1, Test2

a=1	a=0	b=1	b=0	c=1	c=0	d=1	d=0
3	0	2	1	2	1	2	1

Window_1, Test3

a=1	a=0	b=1	b=0	c=1	c=0	d=1	d=0
2	1	1	2	2	1	1	2

Fig. 10 Window_1 feature vectors for Test1, Test2 & Test3

2 cycles. Mapper_1 all trace cycles belong to window_1, accordingly it will emit "window_1" as the key for all its (key, value) pairs as shown in Fig. 9. Mapper_1 counts for every design signal how many times their value changes to "1'b1" or "1'b0" within window_1, accordingly for signal "a" it emits {window_1, (a = 1), 2}.

The map step is followed by a shuffle step that Hadoop does automatically to sort and consolidate the intermediate data from all mappers and before reduce task start. During this step every (key, value) pair is assigned a computing node such that all the occurrence of the signal changes for window (w_i) lands on the same machine. In Fig. 9, during the shuffle phase all the (key, value) pairs that have "window_1" as their key is assigned the 1st node, while the pairs with "window_2" key land on Node2. The final step is the Reduce step, where the aggregation sum is done to sum up all the signal value changes within the trace window (w_i) that lands on the same node during the shuffle phase.

In order to minimize the time required to read the trace dump files LZO compression algorithm [25] is used to shrink their sizes for better disk storage utilization faster processing time. Hadoop supports multiple compression format like lzo, gzip or bzip. LZO is a lossless algorithm for data compression that ensures high decompression speed. Additionally, the block structure of LZO allows the file splitting during the Mapper-Reducer execution. Processing files in LZO format during Map-Reduce function is straightforward. LZO packages are installed in Hadoop. Hadoop configuration is adjusted to include LZO compression codecs. When the mapper wants to process a compressed file, it will check the compression codec and use the suitable codec from there to read the file. Same as for the Reducer Operation. Additionally to share the trace header, which lists the design signal list (a, b, c and d) in our example, between the mappers. Our work flow proposes a solution where the header is saved in Hadoop Distributed File System (HDFS), then a separate code would run before the actual mapper job to serialize it in a file for the mapper to read it during feature extraction. This is required because each mapper would be running alone on a Java Virtual Machine.

After feature extraction, our workflow creates a clustering model for every trace window (*i*). The bug localization algorithm starts by splitting the data into training data and testing datasets. For each trace windows (*i*) across the training dataset, X-means Clustering model [26] is used to group similar trace windows of (*i*) into the same cluster. X-means 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 [26]. Euclidean distance, Eq. 2, is the similarity metric that is used within our K-means clustering algorithm.

$$\sum_{i=1}^{D} X_q[i].X_k[i] \tag{2}$$

whereX_q, X_k are the feature vectors for two input data points. For example, if the following two vectors are window_1 feature vectors for "Test1, Test2". Then the Euclidian distance between "Test1, Test2" is equal to $(9+0+4+1+6+0+4+1)=25$.

	a=1	a=0	b=1	b=0	c=1	c=0	d=1	d=0
Window_1, Test1	3	0	2	1	3	0	2	1

	a=1	a=0	b=1	b=0	c=1	c=0	d=1	d=0
Window_1, Test2	3	0	2	1	2	1	2	1

Once the clustering modules are trained. They can be used to predict for any new test whether its trace windows belong to any of the trained clusters or not. If the test trace window fails to be assigned to any of the trained clusters, it is identified as a buggy trace window. One Nearest-Neighbor Classifier [27] is then used to assign the buggy trace window to the class of its closest neighbor in the feature space. The list of suspected defective design signals is detected by differing the buggy trace window with the nearest neighbor trace window. Our method reports the design module name, the signal's list and the trace window number for the observed buggy behavior.

4.4 Trace Debugger Module

The main goal of Trace Debugger Module is to group similar trace chunks into the same cluster so that the debugging effort can be scheduled for each cluster accordingly. Clusters with a lot of elements indicate redundant trace chunks, and the verification engineer could pick up a few of them for inspection. Clusters with few members reflect trace chunks with unique or rarely occurring behavior so deeper debugging effort should be assigned to them. The input to The Trace Debugger Module is a list of trace windows encoded as a sequence of events. A trace event "E_i" can be defined as the combination of the signal value changes at the clock rising or falling edge.

Table 1 Example for trace dump file

Time	Night	Sensor	p_red	p_grn	s_red	s_yel	Curstate
T1	0	0	0	1	1	0	000010
T2	0	0	0	1	1	0	000010
T3	0	0	0	1	1	0	000010
T4	0	0	0	0	1	0	000100
T5	0	0	0	0	1	0	000100
T6	0	0	0	0	1	0	000100
T7	0	0	0	0	1	0	000100
T8	0	0	1	0	1	0	001000
T9	0	0	1	0	0	0	010000

Accordingly a sequence of events is an ordered list of trace events separated by the time difference (Δ) between them.

For example, Table 1, represents a simple trace dump in which the signal values combination at time step "T1" is "night=0, sensor=0, p_red=0, p_grn=1, s_red=1, s_yel=0 and curstate=000010" and will be encoded as E1. In the trace dump example, E1 sustains for three clock cycles then a new event,E2, is encountered at T4 with the values of "night=0, sensor=0, p_red=0, p_grn=0, s_red=1, s_yel=0 and curstate=000100". This encoding process continues until the entire trace dump is encoded as a sequence of events. The example trace dump ends up with the following encoding scheme "E1 #3 E2 #4 E3 #1 E4 #1 E5".

The input to the clustering-based Trace Debugger module is a list of trace windows "w_i's" presented as a sequence of events, Fig. 11. At the beginning of the machine learning step the encoded event sequences should be projected into the learning space. Our approach utilizes TF-IDF as a feature weighting function to assign higher weights to the most important events in the trace dump. TF-IDF [28] is a common weighting practice in the field of information retrieval and text feature extraction, it emphasizes on the common events that frequently occur locally within the trace segment but rarely in the global entire simulation trace. It is calculated as:

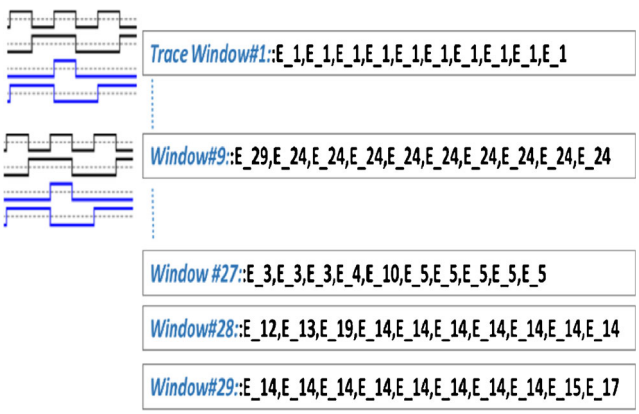

Fig. 11 Trace windows encoding into events sequences

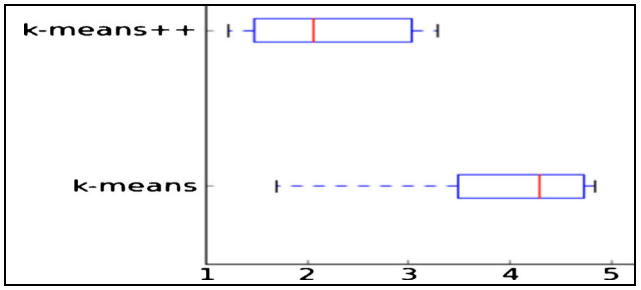

Fig. 12 Clusters Heterogeneities with & without k-means++ initialization

$$TF_{IDF(E_i)} = (\text{No.of times } Ei \text{ appears in i Trace Segment}) * \left(\log \frac{\text{Total No.of Trace segments}}{1 + \text{No.of Trace segments that includes Ei}}\right)$$

The trace debugger module uses a variation of the standard k-means clustering algorithm known as k-means++ [29] for the initial centroids selection. K-means++ spreads out the initial cluster centers by randomly selecting the first cluster center after that every subsequent cluster center is chosen with probability proportional to its square distance from the data points that are closest to the existing cluster center. Accordingly, the next cluster center selection will always be biased to be far from the existing clusters centers. Euclidean distance with normalization, which is estimated cosine similarity [29], is used as the distance metric between cluster centroid and the input observations. The output is a set of cluster assignment to each trace segment.

The quality of the cluster identification and cluster element assignment is calculated using cluster heterogeneity metric [18]. K-means method minimizes the sum of the squared distances between the cluster centroid and the input data points. Cluster heterogeneity is mainly based on summing over the square of all the distances between clusters centroids and their assigned data elements. The smaller the distance the more tight and homogeneous the clusters are. Cluster heterogeneity is calculated as follows:

$$\sum_{j=1}^{k} \sum_{i=1}^{z} \left\| X_i - \mu_j \right\|^2 \tag{3}$$

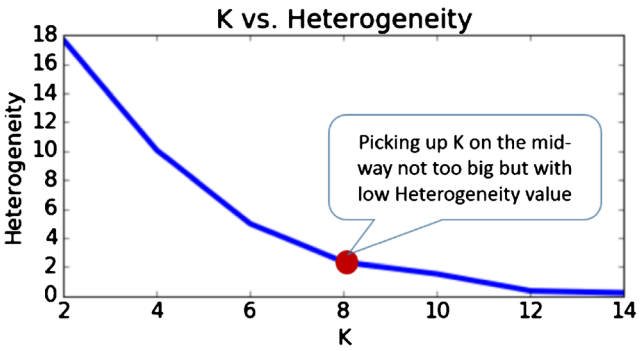

Fig. 13 K-means iterations for best K value selection

Table 2 Training test characteristics

Test Class Name	Number of Training Tests	Trace Dump Size in GBs	Test Length in Cycles
Ethmac_Register_Access_Test	100	0.004	1, 167
Ethmac_Registre_Reset_Test	150	0.04	13,071
Ethmac_TX_Packet_Test	300	4	1,509,759
Ethmac_Tx_CRC_Test	300	3	1,021,719
Ethmac_Rand_RxTx_Test	700	5.5	2,000,000

Where X_i's are the input data points at cluster j and μ_j is the j^{th} cluster centroid. To understand the impact of applying k-means++ for initial centroids identification. Figure 12, compares clusters' heterogeneities from seven restarts of k-means using a random initialization compared to seven restarts of k-means using k-means++ as a smart initialization.

K-means based clustering methods, K-means++, are very sensitive to the selection of correct cluster number "K". Picking up a large number of clusters during the problem solving will produce finer clusters with low heterogeneity values but will cause model over-fitting. To solve this problem multiple k-means runs with different values of k are done followed by clusters heterogeneities calculations for every run. The k values are then plotted with respect to the obtained heterogeneities and k is picked in the midway, i.e. not too big but also has low heterogeneity value. Figure 13 demonstrates the experiment of k-selection step. Trace debugger operation ends up by assigning cluster-ID to every trace window in the execution trace dump.

5 Experiemntal Results

During this experiment, our work is exercised using a group of HW designs downloadable from [30]. It starts by evaluating Bug Localization Module with Ethernet MAC IP Core example using its regression suite of 1500 tests. Then we demonstrate how Trace Debugger Module manages to identify clusters of similar trace chunks for a number of designs such as DRAM controller, AHB-Wishbone Bridge, Handshake module and UART design.

5.1 Bug Localization Experimental Results

We evaluated the Bug Localization Module using Ethernet IP Core [30], which is a MAC (Media Access Controller) connected to the Ethernet PHY chip on one side and to the Wishbone SoC bus on the other. The Bug Localization Clustering Model was trained using 1500 tests from the Ethernet five main test classes, namely, Register Access, Register Reset, Transmitter Packet, Transmitter CRC and Transmitter-Receiver. These 1500 tests were generated by feeding different seeds to the Ethernet constraint random System Verilog testbench environment. The test lengths ranged from about 1167 to 2 million cycles. The size of trace dumps varying from 40MBs up-to 5.5 GBs. Table 2, sums up the test characteristics and their distribution among the five Ethernet test classes.

The Bug Localization Module is evaluated in detecting a set of injected functional bugs where the design logic was modified. The injected bugs are simulated to understand their impact on the design behavior. The RTL design modifications are done using mutation-based-technique [31]. Where a set of faulty versions of the original RTL is created using mutation operators as defined in [32] and listed in Table 3. A group of 60 tests with multiple injected errors is generated across the different test classes. These tests include bug-free ones to exercise the ability of the bug-localizer module to detect true negative results.

The experiment starts by applying the signal selection algorithm for the Ethmac signals as explained in Section 4.1.

Figure 14a, demonstrates the connectivity graph for Ethmac design signals (2342 signals). Each node (circle) represents a design signal and the edge between two nodes represents the direct influence path from the source node to the destination node. Figure 14b, plots the connectivity graph

Table 3 Operators replacement by mutation testing

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

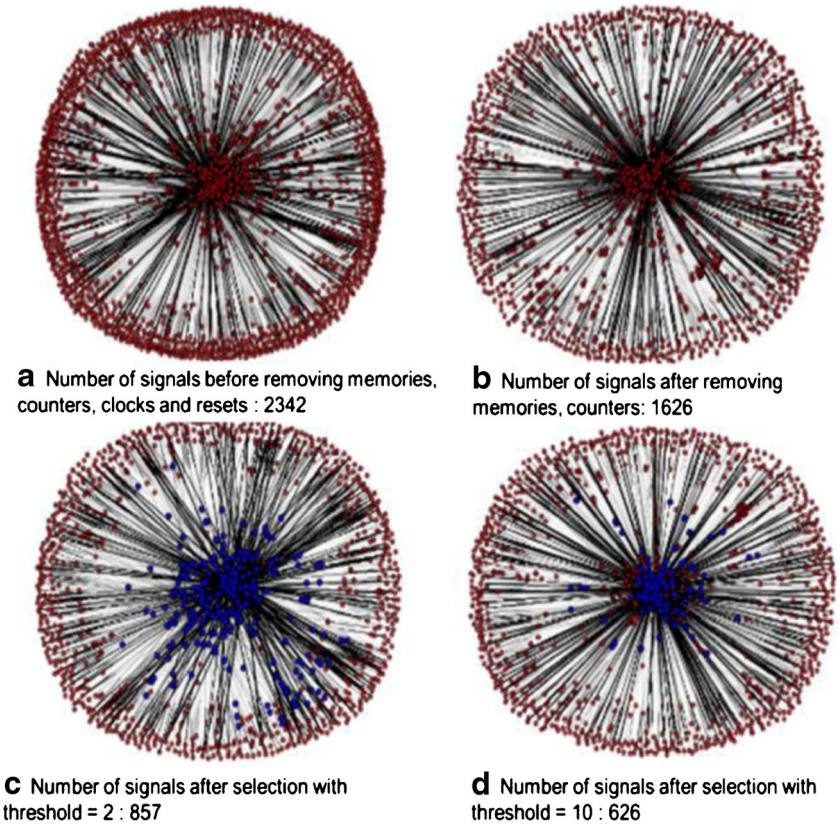

Fig. 14 Impact of signal selection threshold value selection

after statically identifying and pruning out memory states, counters, clocks and resets. This static analysis of the RTL helps in reducing the number of the signals in the design connectivity graph to 1626 signals. Next, signal selection algorithm iterates over the network graph nodes to count every node children as a node influence score. A threshold value of the influence score is defined to suppress the node from the selected signal list, if the node influence score is less than the threshold value. An initial 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 14c, d illustrate ethmac connectivity graphs with threshold values of 2 and 10 respectively. It indicates the impact of the iterative adjustment to the COI score threshold on decreasing the number of the final selected signals.

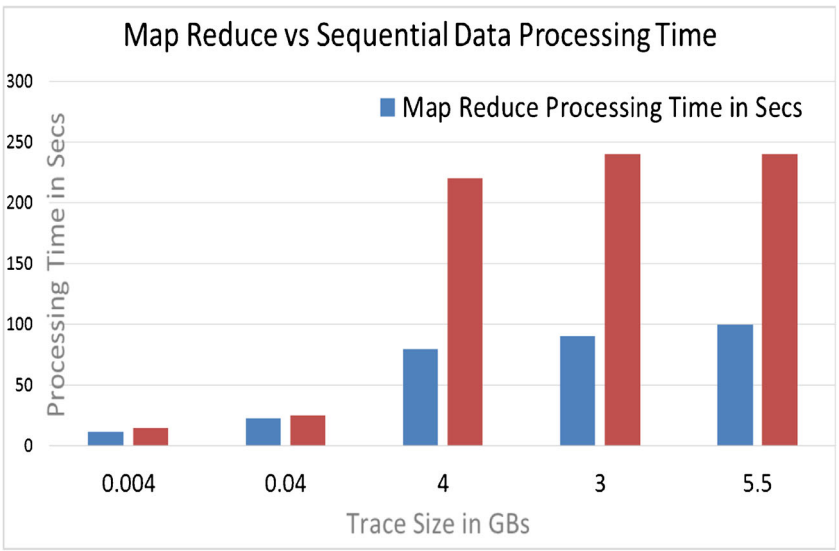

Fig. 15 Run-time for trace dump processing and feature extraction steps with/without map-reduce

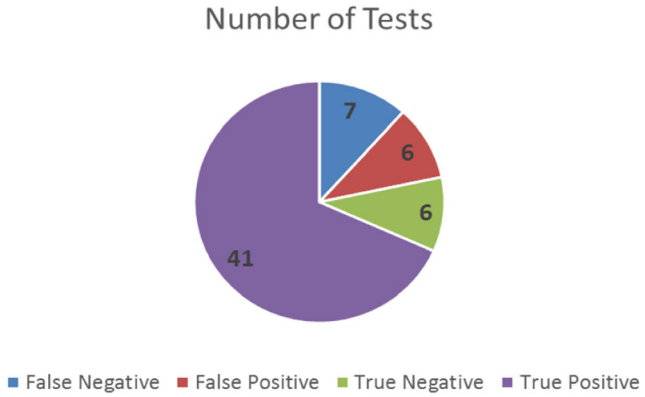

Fig. 16 Bug localizer results categories

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 15, 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 15, demonstrates that the Map-Reduce accelerates the trace dump encoding as well as feature extraction steps by a factor of 2.7× faster than the sequential data processing techniques.

The effectiveness of Bug Localization Module is evaluated using three metrics namely, accuracy, the bug detection latency and the number of design signals impacted by bug injection versus the number of detected signals by the localizer module.

Model accuracy reflects the ability of the model to make correct predictions. It is defined by capturing four categories of model results:

- ***False Positives:*** occur when a bug is detected, although there are no injected bugs
- ***False Negatives:*** occur when the model misses the detection of a real injected bug.
- ***True Positive:*** occur when an injected bug is detected
- ***True Negative:*** occur when no bug injected and no one detected as well.

Figure 16, sums up the results obtained by evaluating the Bug Localization clustering model using a set of 60 buggy tests. It indicates accuracy of 78% where the model manages to detect true injected bugs (True Positives) as well as to identify no bugs when there are no injected bugs (True Negatives). It also demonstrates that for 10% of the cases the localizer detected bugs when there were no injected bugs (False Positive). And in 12% of the cases it missed the detection of truly injected bugs (False Negative).

Another evaluation metric for the Bug Localization Module is the bug detection latency. The latency is the number of cycles from bug injection to bug detection. Figure 17, shows bug detection latencies across the test cases. It shows in 68.33% of the tests, bugs were detected at the same cycle of bug injections. In ~19% of the tests, bugs were detected quickly, within 5 to 10 cycles from their injection sources. While in 8% of the tests, the bugs were detected after longer time (> 20 cycles) from their injection time. Finally, only 5% of the test cases the bugs were not detected at all. This happens when the bug's accumulation impact on the design's behavior becomes un-detectable. On average, our system was able to detect bugs very quickly after injection only 12 cycles, averaged over all bugs. Recall that bug injection done through mutation of logical operators in the design RTL, so we do not expect very large latencies between bug injection and bug detection cycles.

The number of the detected buggy signals by the clustering module also has an impact on the debugging process effectiveness. Accordingly, we define another evaluation metric to the Bug localization Module, which is the actual number of design signals affected by bug injection vs the observed number of buggy design signals during design execution. Recall that, logical-operator mutations at the RTL injects some defects in a group of signals and correspondingly their secondary

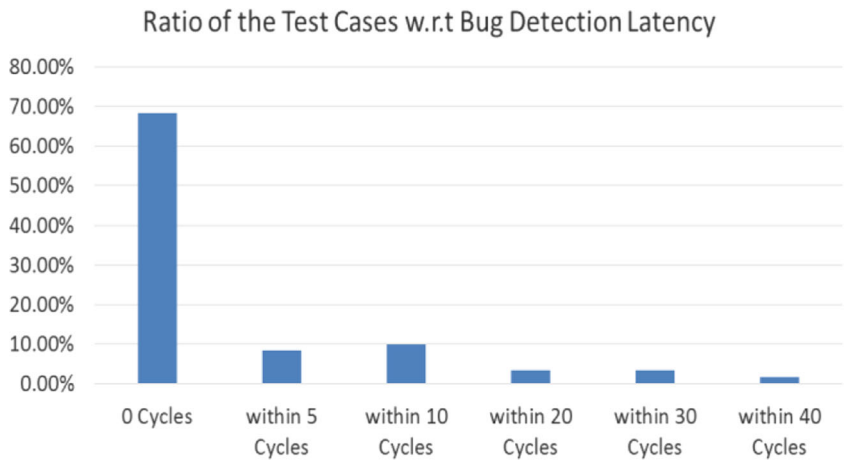

Fig. 17 Distribution of tests bug detection latencies

effects may impact other design signals. While, some other injected defects may not even propagate to any of the observable debugging signals. Figure 18, demonstrates for each buggy test case the actual number of defective signal vs the number of detected buggy signals. Figure 18a, indicates that with 65% of our tests (39 tests) the number of actual signals that are affected by injected bugs match the number of detected signals by the localizer module. While, in 23% of tests (14 tests) the Bug-Localization Module detects more number of buggy signals than the actual impacted signals by the injected bugs. This is considered as a partial success, Fig. 18b. Finally, in Fig. 18c, 10% of tests, the localizer module does not detect any buggy signals or detect less number than the injected defective signals.

We conclude this subsection by a comparison between the Bug Localization Module and the previous work at [8], Table 4. Firstly, the work in [8] utilizes a mix between unsupervised machine learning anomaly detection methods and K-Means Clustering. Where the bug time and signal localization are done in two separate steps. A pass and fail labels should be available with the training data. Our work combines the bug time and signal localization in one step where only unsupervised clustering using X-Means technique is used to automatically determine the number of clusters and hence avoid this limitation with k-means clustering technique. Secondly, we introduce a signal selection algorithm that control which signals to log during design execution on the FPGA Prototype. Our signal selection algorithm depends on the design static analysis to identify COI of the selected signals. The work in [8] selects a limited number of signals that span across the different design modules for post-silicon debug. Moreover, the work in [8] learns the good behavior of the design by multiple executions of the same test when some of these trials may pass while others fail. Our approach learns the correct design behavior by training the machine learning algorithm across the entire design regression test cases with its different test classes. Additionally, our work introduces for the first time the utilization of Map-Reduce method to accelerate the processing of the huge resulted execution trace from design prototyping hence accelerate the feature encoding step with 3× compared to the work in [8]. Our work improves the feature encoding step by considering the frequency of signal's value change to 1'b0 and 1'b1 within every trace window. Both approaches depend on dividing the trace dump into trace windows of fixed size. Our approach uses a finer trace window size results in better bug detection latency, i.e. ability to detect bugs closer to their root cause. Our Bug Localization Module reports 78% accuracy which is 25% better than the work presented in [8]. Finally, our work flow focuses only on the function bugs, while [8] works with electrical and manufacturing bugs as well.

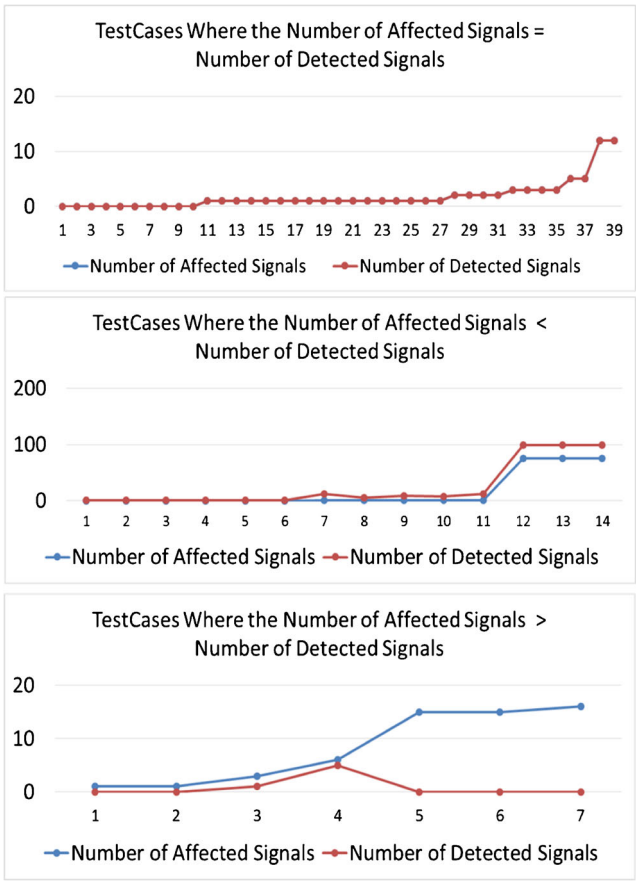

Fig. 18 **a** in 65% of tests the number of detected defective signals equal to actual number of impacted signals, **b** in 23% of tests the number of detected defective signals is greater than the actual number of impacted signals, while in (**c**), 10% of the tests have the Number of detected signals is less than the number of impacted signals

5.2 Trace Debugger Experimental Results

During this experiment a group of HW designs is exercised with the Trace Debugger Module. The main goal of the Trace Debugger Module is to help the verification engineer understanding more about the trace under debugging. By pointing to the redundancies in the long trace file and highlighting the trace parts where unique behaviors occur. The Trace Debugger Module operation starts by dividing the trace file into trace segments/Windows. Where every trace window is encoded as a sequence of events. Table 5, lists for every design the number of events encountered in its trace dump, the number of its trace segments and the number of events within every segment.

In order to evaluate the effectiveness of Trace Debugger Module, we need to understand more about the redundancy inherited in every design trace dump. Accordingly, we define a *Redundancy Ratio* as a metric that indicates the ratio of the identical trace segments over the entire trace segments. Figure 19, demonstrates the redundancy ratio in each design trace dump. This gives an indication about the number of repetitions in the trace dump of every test case. For example

Table 4 Comparison between bug localization module and reference work at [8]

Anomaly detection [8]	Proposed bug localization module	
Machine learning methods	Supervised anomaly detection where both passing & failing labels are required for training data, clustering- based anomaly detection	Unsupervised clustering with no-label associated to training data
Training data	Multiple execution of the same test	Design regression tests across different test classes
Features extraction	Fraction of time the signal value is 1'b1 during trace window	Frequency of signal's value changes to 1'b0, 1'b1 during trace window
Bug time & signal detection	Time localization & signal localization are done in two separate iterations	Time & signal localization are done in same step
Bug types	Function, electrical or manufacturing	Function only
Design description level	RTL or trace dump availability	RTL or trace dump availability
Signal selection for debugging	Select a group of signals to cover different design modules	Formal cone of influence signal selection algorithm
Time Window Cycles	500 Cycles	Fine grained trace windows with 100 Cycles
Feature extraction speed	Sequential features encoding	~3× Faster with Big-Data Map Reduce
Bug localization latency	Within trace window limit Avg. of 60% of trace window size after Bug injection (Avg. of 300 cycles within 500 cycles trace window)	Within trace window limit Avg. of 12% of trace window size After bug injection (Avg. of 12 cycles within 100 cycles trace window)
Accuracy	53%	78%
False positive, False negative	18%, 29% respectively	10%, 12%

WB_2_DMA, DRAM_Controller, UART_APB_Protocol and UART_TX_FIFO designs have very few unique segments in their trace dump so they have a huge redundancy in the recorded design execution traces. While the Interleaver, Robot_Controller, UART_RX_FIFO and UART_Transmitter_Channel designs have few similarities between their trace segments as shown in Fig. 19.

K-means clustering is used to group similar trace segments into the same cluster. Our algorithm utilizes K-means++ for the smart initialization of cluster centroids at the first iteration.

Table 5 Trace debugging module tests characteristics

Design name	Number of trace events	Number of trace segments	Number of events per trace segment
AHB-to-WISHBONE Bridge	220	43	5
WB_2_DMA	154,021	1817	85
Interleaver	14,240	712	20
Robort_Controller	5000	167	30
DRAM_Controller	5866	587	10
UART-Controller	43,160	864	50
UART_APB	43,160	864	50
UART_APB_Protocol_ Monitor	43,160	864	50
UART_Receiver_Channel	43,160	864	50
UART_RX_FIFO	43,160	864	50
UART_Transmitter_Channel	43,160	864	50
UART_TX_FIFO	43,160	864	50

The most appropriate number of cluster count "k" is selected after multiple k-means iterations to compromise the tradeoff between over-fitting and cluster quality measured by heterogeneity as discussed in Section 4.4. The output of the trace debugger module is a Cluster ID assignment to every trace segment. To demonstrate visually the output of the trace debugger module, we exercise the module with a simple traffic light controller design, and presents the results in Fig. 20. Figure 20, displays some examples for trace segment assignment to cluster_1, cluster2 and cluster_3. With a closer look to the trace waveforms, it is clear how the trace segments that are belonging to the same cluster reflects a redundant trace behavior in the execution trace.

Table 6, sums up the Trace Debugger results with our designs list. It lists for every test case the number of generated

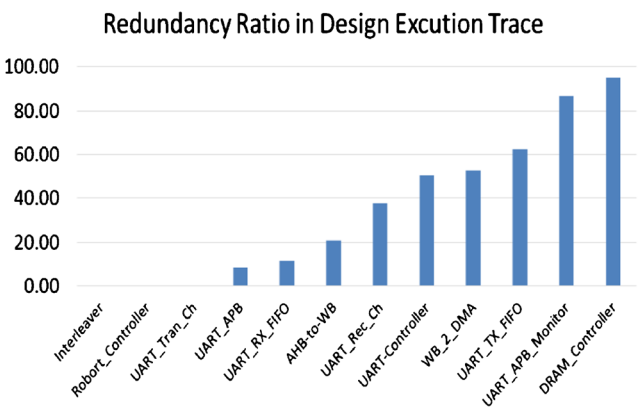

Fig. 19 Ratio of redundant events in test trace files

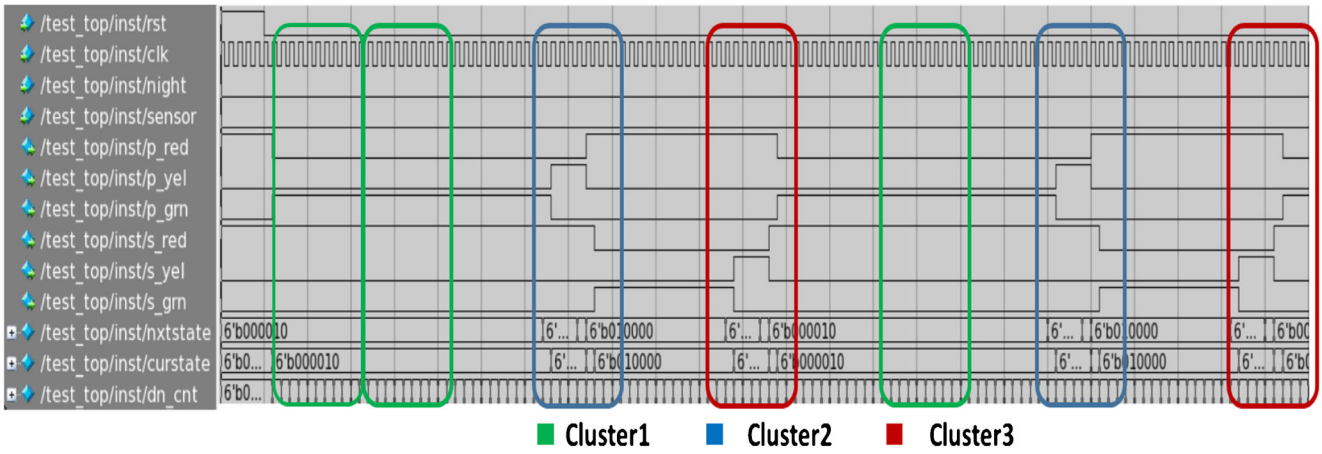

Fig. 20 Traffic_lc example of trace segments to cluster IDs assignment from trace debugging module

clusters, the number of trace segments assigned to every cluster (number of cluster members) and the heterogeneity measure of the resulted clusters. The results in Table 6, indicate that for the HW designs with high redundancy ratio (DRAM_Controller, UART_APB_Monitor, WB_2_DMA, UART_TX_FIFO) the clustering algorithm ends up with a small number of clusters containing large number of trace segments members and small heterogeneity value. This means that the clustering algorithm succeeds in grouping these similar trace segments in a few clusters. The generated cluster members are tightly related with a small distance between them and their cluster centroids (low heterogeneity Value). On the other side, for the test cases with very low redundancy ratio (Interleaver, UART_APB, UART_Controller, Robot_Controller and UART_RX_FIFO), the clusters heterogeneities are relatively high. Additionally, the clustering algorithm ends with large number of clusters having very few cluster members. This proves that the trace debugger module failed to find a good grouping for these dissimilar trace chunks. Accordingly, the clustering algorithm ends up with a large number of clusters, containing few members that are scattered and have large distance between each other's as well as their cluster centroids (high heterogeneity value).

5.3 Map-Reduce with Data Processing & Feature Extraction Benchmarks

In this subsection, we give some insights about how Map-Reduce helps accelerating trace data processing and feature extraction for the ML-Based debugging modules. It demonstrates how Big-Data processing using Map-Reduce gives us the ability to hone into the clustering problem faster and accelerates giving the relevant data to the ML-Based modules.

To study the impact of Map-Reduce in accelerating trace data processing and feature extraction steps for Bug-Localization-Module. We exercised a group of tests for each design in which we extended the execution time to obtain big-sized trace files with a large number of trace cycles.

Table 6 Trace debugger clustering results: clusters assignments, cluster heterogeneity w.r.t trace redundancy

Design name	Redundancy Ratio In Design Excution Trace	Total Number Of Generated Clusters	Avg. no of segments within the cluster	Clusters heterogeneity
AHB-to-WISHBONE Bridge	20.93	16	3	0.89
WB_2_DMA	52.56	10	181	12.65
Interleaver	0.00	30	24	668.10
Robort_Controller	0.00	20	8	146.99
DRAM_Controller	95.06	12	48	18.78
UART-Controller	50.58	60	14	57.57
UART_APB	8.33	100	9	79.18
UART_APB_Protocol_Monitor	86.69	10	84	0.73
UART_Receiver_Channel	37.85	500	2	269.00
UART_RX_FIFO	11.57	70	12	687.30
UART_Transmitter_Channel	0.00	300	3	237.36
UART_TX_FIFO	62.38	150	6	23.58

Table 7 Map-reduce vs sequential data processing & feature extraction for a group of tests

TestName	Trace file size in GB	Trace length	Number of trace signals	Map-reduce time in (Secs)	Time in (Secs) w/o map reduce
Traffic_Test1	0.907	6,172,428	25	711	2212
Traffic_Test2	8.45	61,724,156	25	2922	36,212
AHB-to-WB-Bridge_Test1	3.8	10,726,476	47	940	2132
AHB-to-WB-Bridge_Test2	35	107,260,476	47	4466	33,935
DRAM-Controller_Test1	0.335	990,825	48	40	52
DRAM-Controller_Test2	32.6	96,298,062	48	3815	42,016
FIFO-RAM-Test1	1.6	4,000,707	55	67	112
FIFO-RAM-Test2	16	40,000,707	55	316	1579
UART-Test1	0.101	19,875	171	20	21
UART-Test2	1.43	238,280	371	121	243
UART-Test3	0.212	41,721	171	35	40
UART-Test5	10.1	1,561,033	371	832	7892
UART-Test4	100	21,209,350	171	3452	37,995
Ethernet-Test1	0.004	1167	626	13	15
Ethernet-Test2	0.04	13,071	626	23	25
Ethernet-Test3	3	1,021,719	626	90	240
Ethernet-Test4	4	1,509,759	626	80	220
Ethernet-Test5	5.5	2,000,000	626	100	240

Table 7, lists for every test: trace size in GBs, trace length in cycles and the time for trace encoding and feature extraction using Map-Reduce vs normal sequential data processing methods. We observed that the amount of speedup obtained from the application of Map-Reduce depends heavily on the trace size, the trace length (Number of Cycles), the number of design signals that are observed during design execution and the frequency of signal value changes during the design execution.

Figure 21, plots the obtained run-time improvement ratio between normal sequential data processing vs Map-Reduce method. It demonstrates that for all cases the MapReduce beats the traditional sequential data processing for trace dump and feature extraction runtime. The MapReduce impact is small with the tests where the trace dump size is very small. This is because the overhead of creating parallel mappers for many small-sized trace chunks is larger than the speed-up gain from the parallel operation of the map-reduce. Generally, Hadoop creates separate mapper for each DB block for its HDFS operations, so for many small-sized files the speed up gain will be small (~1.3 in UART Test1/Test2, Ethernet Test1,2,3,…). However with the tests that are having large trace size and long trace cycles the impact of MapReduce in accelerating the processing time is really high with speedup improvement that reaches in some cases up-to 10× (DRAM-Controller Test2, UART Test4/Test5).

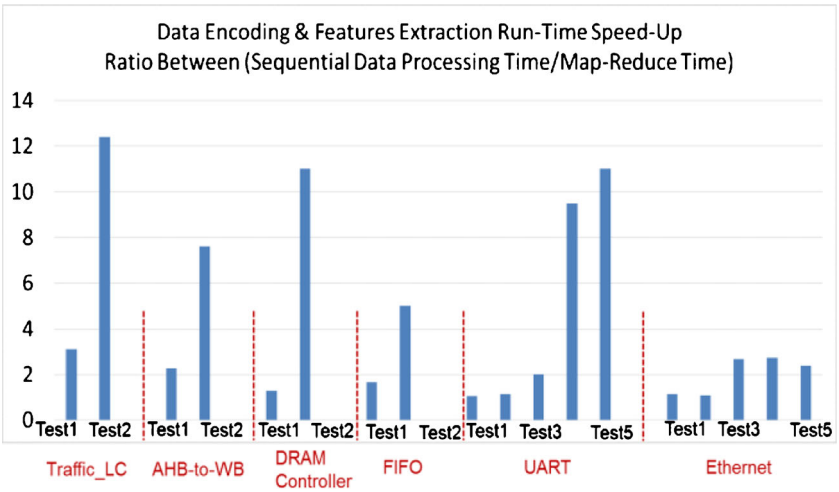

Fig. 21 Speedup in data encoding & feature extraction using map-reduce vs sequential data processing methods

6 Conclusion

This paper presents a framework that facilities post-silicon validation. It utilizes machine learning methods for off-line analysis of the recorded design behavior during design prototyping. Our work proposes a bug localization solution that can identify the buggy trace segment time associated with the design module and the signals related to the detected bug using clustering machine learning. The results indicate that the bug-localization module manages to detect the injected bug with up to 78% accuracy. Compared against other state-of-art solutions in this space, our Bug Localization module can detect injected defects with 25% improvement. The proposed framework uses Big-Data techniques, namely Map-Reduce to accelerate trace data encoding and feature extraction before machine learning step. We demonstrate an average of 4× speed up using Map-Reduce versus traditional sequential data processing methods. A signal selection algorithm based on formal COI analysis is introduced. This helps in identifying which subset of design signals to incorporate during feature extraction step and hence during post-silicon debugging. Our results indicate that in 90% of the tests the selected signals for debugging were able to manifest the bug injected at the design RTL.

We also introduce an approach to guide the planning and scheduling of trace dump debugging by using-clustering based technique to group every similar trace chunks into same clusters that can be debugged by verification engineers one at a time. It also points to trace chunks that owns unique design behavior for deeper debugging. We illustrate for the first time, how the trace dump windows can be encoded as event sequences using TF-IDF method. For the Trace Debugger module, the result is very interesting. It indicates how the proposed method can help the verification engineer suppress the huge redundancy inherited within long trace dumps and upload the trace periods which own unique design behaviors for deeper debugging effort. Our experimental results demonstrate how the proposed method manages to detect few tight clusters with a large number of elements for the HW designs with high redundancy ratio in their execution trace chunks. It also proves that for designs with low redundancy ratio, our approach fails to group these un-similar segments in a few clusters, hence it generates a large number of clusters with few cluster elements that are scattered from each other and from their cluster centroids.

References

1. S. Mitra et al., Post-silicon validation opportunities, challenges and recent advances, In Design Automation Conference, DAC, 2010
2. M. Dehbashi, A. Sülflow, and G. Fey, Automated design debugging in a testbench-based verification environment, In Euromicro Conference on Digital System Design (DSD), 2011
3. E. Singh, C. Barrett, S. Mitra, E-QED: electrical bug localization during post-silicon validation enabled by quick error detection and formal methods, In Computer aided verification, CAV 2017, pp. 104–125
4. S. Park, A. Bracy, H. Wang, S. Mitra, BLoG: post-silicon bug localization in processors using bug localization graphs, In Design Automation Conference, DAC 2010
5. A. DeOrio, D. S. Khudia, V. Bertacco, Post-silicon bug diagnosis with inconsistent executions, In the proceedings of IC Computer Aid Design ICCAD, 2011
6. C. Richard Ho, M. Theobald, B. Batson, J.P. Grossman, Post-silicon debug using formal verification waypoints, In the proceedings of the Design and Verification Conference, DVCon, 2009, PP1–7
7. M. Dehbashi, G. Fey, Automated Post-Silicon Debugging of Design Bugs, http://www.informatik.uni-bremen.de/agra/doc/konf/11S4D_APSDDB.pdf
8. 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
9. X. Liu and Q. Xu, Trace signal selection for visibility enhancement in post-silicon validation, In the Proceedings of Design Automation and Test in Europe, DATE, 2009, pp. 1338–1343
10. Ko HF, Nicolici N (2009) Algorithms for state restoration and trace-signal selection for data acquisition in silicon debug. IEEE Transactions Computer Aided Design Integration Circuits System 28(2):285–297
11. Basu K, Mishra P (2013) RATS: restoration-aware trace signal selection for post-silicon validation. IEEE Transaction Very Large Scale Integration (VLSI) System 21(4):605–613
12. D. Chatterjee, C. McCarter, and V. Bertacco, Simulation-based signal selection for state restoration in silicon debug, In the Proceedings of IEEE/ACM International Conference Computer-Aided Design (ICCAD), 2011, pp. 595–601
13. K. Rahmani, P. Mishra, and S. Ray, Efficient trace signal selection using augmentation and ILP techniques", International Symposium on Quality Electronic Design (ISQED), 2014, pp. 148–155
14. S. Prabhakar and M. Hsiao (2009) Using non-trivial logic implications for trace buffer-based silicon debug. In: Asian Test Symposium, ATS. pp. 131–136
15. Li M, Davoodi A (2014) A hybrid approach for fast and accurate trace signal selection for post-silicon debug. IEEE Transactions Computer-Aided Design of Integrated Circuits and Systems 33(7):1081–1094
16. H. F. Ko and N. Nicolici, Combining scan and trace buffers for enhancing real-time observability in post-silicon debugging, In the Proceedings of 15th IEEE Europe Test Symposium (ETS), 2010, pp. 62–67
17. K. Rahmani and P. Mishra, Efficient signal selection using finegrained combination of scan and trace buffers, In Proceedings of 26th international conference VLSI design, 2013,pp. 308–313
18. H. F. Ko and N. Nicolici, Automated trace signals selection using the RTL descriptions, In Proceedings international test conference, 2011, pp. 1–10
19. E. Fox, C. Guestrin, Machine Learning: Clustering and Retrieval, https://www.coursera.org/learn/ml-clustering-and-retrieval/home, 2016
20. Questa Formal Property Check, https://www.mentor.com/products/fv/questa-property-checking
21. Networkx,"A software for complex network", https://networkx.github.io/
22. Certus Silicon Debug, https://www.mentor.com/products/fv/certus-silicon-debug
23. 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
24. T. White, "Hadoop: The Definitive Guide", 2015, 4th Edition, ISBN: 978-1-491-90163-2

25. A guide to using LZO compression in Hadoop, 2012, https://nnc3.com/mags/LJ_1994-2014/LJ/220/11186.html
26. 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
27. M. Narasimha, V. Susheela “Pattern Recognition, An Algorithmic Based Approach”, 2011,Springer 978–0–85729-495-1, pp 48–85
28. Manning CD, Raghav P, Schultz H (2008) Introduction to information retrieval. Cambridge University Press, ISBN 0521865719
29. D. Arthur, S. Vassilvitskii, "k-means++: the advantages of careful seeding”, In the Proceedings of the 18th ACM-SIAM symposium on Discrete algorithms, 2007,PP. 1027–1035
30. Opencres benchmarks “http://opencores.org”
31. 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
32. Mutation operators listing http://pitest.org/quickstart/mutators

Eman El Mandouh is Senior Quality Assurance Manager for IC Verification and Validation System Division, Mentor Graphics Siemens Business. She has more than 18 years of experience in the field of digital design and functional verification technology namely simulation based verification, advanced formal verification solutions, Coverage driven verification, intelligent test generation, FPGA Prototype & Debugging. She is a Ph.D. candidate in Cairo University where she obtained her B.Sc. and M.Sc. in Computer Engineering. Her PHD graduation thesis is about “Exploring Machine Learning techniques in Functional Verification Field”, she has published number of articles in the field of assertion based verification, machine learning with Function Verification and Software testing. Working as quality manager for EDA, Eman has been certified from world quality institutes such as Green Belt Six Sigma for Information Technology and Software Engineering from the European Software Institute (ESI) and Quality Manger and Organizational Excellence from American Society of Quality.

Amr G. Wassal is currently an Associate Professor at the Computer Engineering Department, Cairo University. His current research interests are in the areas of reconfigurable and multi-core DSP processors for wireless and MEMS applications. He also has active interest in different areas of EDA algorithms and technologies, as well as, control architectures for electro-mechanical systems. In the Industrial Field, Amr has 22 years of R&D experience in the semiconductor industry. He started with SWS by establishing and growing the digital design team, and is currently the head of operations and systems engineering. Prior to joining SWS, Amr held several senior managerial and technical positions at PMC-Sierra, ICCC A/S Denmark, and IBM Technology Group. He was also affiliated with the VLSI Research Group and the Center for Applied Research in Cryptography at the University of Waterloo. Amr’s expertise is in the areas of digital and DSP architectures and chip development and productization. He has led teams for over 20 chips targeting highly innovative applications ranging from MEMS-based sensor interfaces to cryptographic co-processors to large storage and telecom SoCs. During his tenure at PMC-Sierra, he was on the T10 Technical Committee at the ANSI INCITS T10 where he made technical contributions to the 6GSAS PHY Working Group. Amr obtained his B.Sc. and M.Sc. degrees in electronics and communications engineering, both with highest honors, from Cairo University. He received his Ph.D. degree in electrical and computer engineering from the University of Waterloo, Canada. Amr is author and co-author of more than 30 publications in international journals and conferences and holds 1 US patent and has 2 more pending.