AI-based Algorithms to Analyze and Optimize Performance Verification Efforts

Saksham Mehra, Raghu Alamuri, Sharada Vajja (Google LLC)

Abstract-- *Mobile System-on-Chip (SoC) devices have entered an era of unprecedented complexity, driven by the exponential growth in hardware capabilities within smartphones and other portable devices. The increased demand to support complex and high performance features in the current smartphones has prompted the integration of compute engines like GPUs, multicore CPUs and other Machine Learning processing units in the mobile chips. This substantially surged the performance demands placed on the mobile devices making performance verification an increasingly critical endeavor in addition to the functional verification. In this paper, we address two critical aspects of efficient and scalable performance verification through our proposed AI algorithm flow. We streamline stimulus generation and optimize the simulation time of a usecase by running only a subset that is representative of the entire usecase. We leverage the same algorithm to analyze and characterize performance failures, significantly expediting the debugging process.*

I Introduction

Performance issues in SoCs often exhibit a cascading effect wherein a throughput bottleneck in one IP at a specific time instance might impact a different IP at a distinct time. This complexity requires a multi-level approach to problem solving from IP level testing to post silicon validation. Performance analysis in the SOC domain gauges resource utilization including but not limited to meeting throughput and latency requirements. The complexity arises due to multiple operating points, multiple clock frequencies, multiple initiator and target interfaces on both datapath and control path and other factors impacting performance like quality of service (QoS), arbitration mechanisms etc.

In this paper, we present an AI-based algorithm to efficiently capture the intricacies of the performance verification flow, offering insights into strategies, methodologies , stimulus generation, analysis techniques and results. More importantly, this approach is designed to be scalable across various testbenches and environments including SoC modeling, IP and SoC level RTL simulations, emulation and post silicon validation.

The rest of the paper is organized as follows, we first define the problem statement in Section II. In Section III, we discuss our proposed solution. Section IV gives a brief introduction to the algorithms used. We discuss our experimental results in Section V. Finally in section VI, we have conclusion and future work.

II Problem Statement

Performance Test Stimulus: Long Duration Use Case-Based Benchmarks

Use case-based benchmarks serve as critical tests in measuring the performance and functionality of SoCs. The ultimate goal of any SoC is to run full benchmarks or use cases to stay competitive in the industry. Running such scenarios on RTL early in the project life cycle enables designers to identify and fix bugs before they escalate into more complex issues. To detect and mitigate bugs early and prevent costly redesigns, RTL simulation platforms are extensively used but are relatively slow in terms of runtime. Long-running simulations of usecases are not feasible to run on RTL simulation platforms due to the runtime limitations. Therefore, an alternative approach is needed to make usecases simulation friendly, yet covering full functionality. AI classification algorithms can be applied to solve this problem.

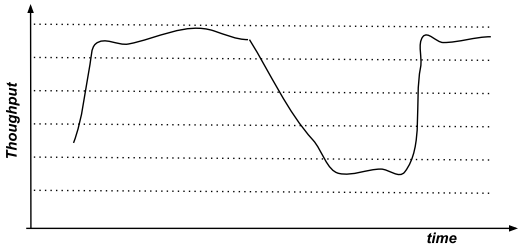
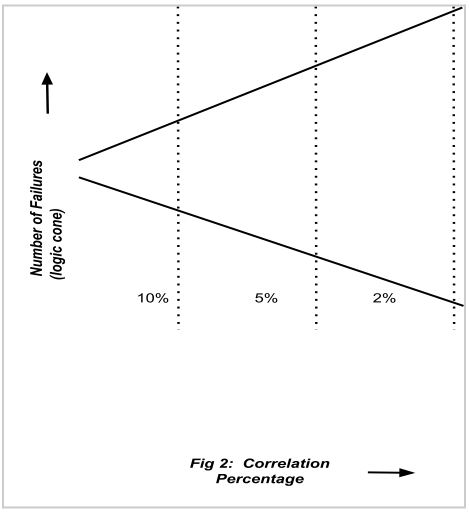
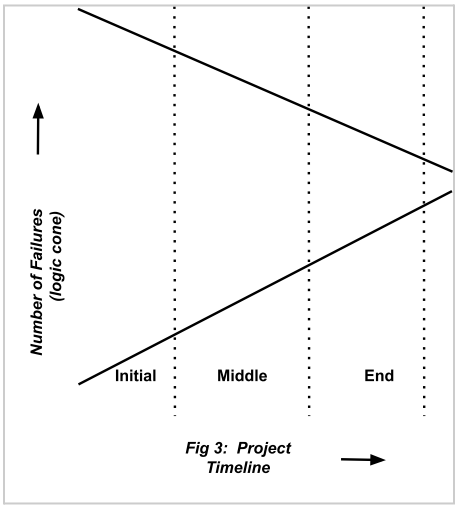


Fig 1: Usecase showing a typical repetitive throughput pattern over time

Performance Test Failures

Performance failures are different from functional failures because they have a single, unique signature: a mismatch in total cycle count. The number of failures will vary depending on the project timeline and how tightly correlated the criteria is kept. As shown in Figure 3 below, the number of failures decreases as the





project gets closer to tapeout. Similarly, the number of failures increases as the correlation percentage target decreases as can be seen in Figure 2. With a correlation criteria of 10% vs 2% there could potentially be more tests that fail. Triaging and prioritizing performance outliers with a single signature is difficult when the number of outliers are large in count. AI classification algorithms can be applied to solve this problem.

III Solution

In order to manage potentially overwhelming performance state space, we have come up with AI-Based methodologies to identify the right and sufficient stimulus (**AI-Based Stimulus Generation**) and outliers to be debugged with high priority (**AI-Based Performance Analysis**). Our proposed methodology leverages popular statistical analysis techniques like principal component analysis (PCA) and clustering

algorithms (K-means clustering algorithms) [5][7]. Our test scenarios are aligned across modeling, simulation and emulation in order to do correlation. We have automated performance verification flow across various environments where the same stimulus file is read in by the model and RTL and tests are run in both. We then compare metrics like latency and throughput for a 3-way correlation across theoretical expectation, model and RTL.

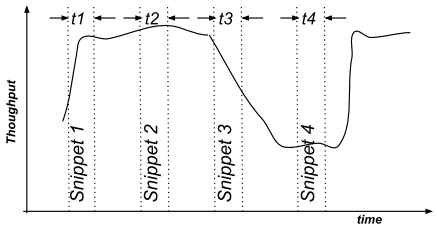


Fig 4: Unique snippets of a repetitive usecase throughput pattern

AI-Based Stimulus Generation

Many use cases have repetitive behavior as can be seen in the example in the plot Figure 1 with repetitive throughput pattern over time. Identifying the traffic pattern helps to divide the full use case into meaningful, simulation-friendly snippets. Figure 4 shows the throughput vs. time behavior of one use case, where the throughput is recurrent in nature which renders it unnecessary to run the entire usecase test in simulation or emulation environment. Throughput alone doesn't fully characterize use case behavior, as it can be influenced by a range of other factors such as cache hits/misses, traffic, address patterns etc. Identifying similar use case behaviors is a challenging task. Therefore, AI-based techniques can be used to identify similar time snippets and characterize the most representative snippets of the entire usecase which can then be replayed in RTL simulation and emulation more effectively. Fig 5 shows the steps involved in our algorithm workflow.

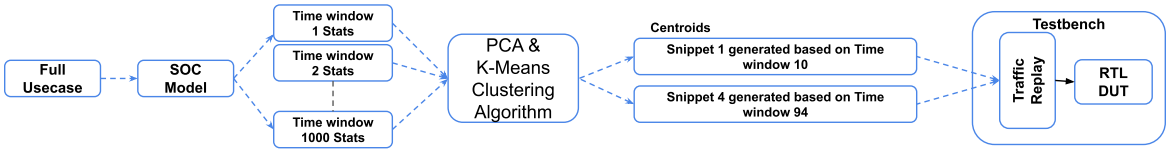


Fig 5: K-means clustering algorithm to identify hotspots

The following are some example statistics that can be attributed to each time window to help AI algorithms classify unique behaviors and form clusters, statistics like *average bandwidth, median latency, outstanding count, DRAM refresh count, DRAM page hits, MMU cache hits, data cache hits etc.* The accuracy of the algorithm progressively improves with the number of statistics used.

AI-Based Performance Analysis (outliers classification)

Analyzing performance verification failures pose a unique challenge. Unlike functional simulations, performance verification failures often lack unique and self descriptive error signatures, most often showing up as cycle count mismatch or throughput discrepancies.

To address this challenge, advanced AI algorithms are deployed to classify failures into distinct buckets thus enabling the identification of critical issues with higher debug priority [4]. Notably, 90% of failures arise from common bottlenecks falling into a "**Big-Bucket**" category. The remaining 10% failures are assigned equal weightage as they may represent different **corner case scenarios** which are critical to debug.

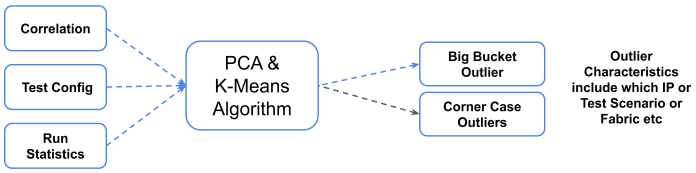


Fig 6: Algorithm Flow for Performance Failures Classification

During the prioritization of failure categories, the AI-based tool considers multiple parameters from the test config and runtime statistics including traffic patterns, average outstanding count, median and structural latencies, back pressure, bandwidth discrepancies and more. These parameters are fed into our algorithm which then automatically classifies the tests into the respective buckets or clusters as shown in Fig 6. The tool expedites the time to debug by associating failures with potential root causes.

IV Algorithms (PCA & K-Means Algorithm)

K-means is an unsupervised machine learning algorithm that groups data into distinct clusters and works by iteratively minimizing intra-cluster variance while maximizing inter-cluster variance. It requires feeding the value of k(number of clusters) and iteratively updates cluster centroids until convergence.

Large dimensionality of input data plays significant challenges in SOC performance verification, as capturing numerous performance metrics for each testcase leads to computational inefficiencies while clustering.

PCA is a vital dimensionality reduction technique that simplifies high dimensional dataset while preserving the crucial information.

We use a very similar flow for both the tasks at hand **(AI-Based Stimulus Generation and AI-Based Performance Analysis).** The approach involves following primary steps :

- To ensure the algorithm's proper functionality the input dataset has to be normalized, failure to perform this critical pre-processing step can lead to disproportionate weightage of features with varying numerical scales, potentially leading to inaccurate outcomes.

- As discussed earlier, we use PCA to reduce the number of features in our data before feeding it into our clustering algorithm [1]. For the optimal number of principal components, we adopt either of the two techniques, namely Cumulative Variance graphs (Fig 13) or Silhouette Score (Fig 14). We establish a selection criterion, targeting 80% (determined experimentally) cumulative variance as the threshold for determining the suitable number of PCA components. Silhouette score [8] plot captures the overall quality of clustering for varying PCA components.

- A crucial step that determines the accuracy of our algorithm implementation is to come up with an optimal number of clusters . Fig 7 shows our plot of within cluster sum of squares (WCSS) against the number of clusters. The underlying concept revolves around the identification of the juncture where further increment in the number of clusters ceases to yield a substantial reduction in the WCSS score. This strategic approach is commonly recognized as the "elbow method" [3] [6].

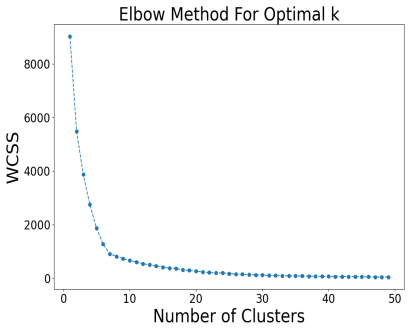


Fig 7: Elbow method curve for optimal clusters

- Once we know the appropriate number of clusters, we feed this information along with the PCA components into our k-means algorithm, which provides centroids of each cluster along with cluster labels for each datapoint. We are more interested in identifying the point central to each cluster which is a representative of all points in the cluster. Our criterion for selecting this point is to find the data point nearest to the mathematical cluster centroid. Depending on the specific circumstances, this

central point may encompass a single data point or, in certain scenarios, a set of data points. Unless otherwise specified, we call this central point a centroid (though mathematically it is not) in our further discussion. The distance metric used is Euclidean distance.

V Results

Test Stimulus: Simulation and Emulation friendly

We ran a complete single initiator performance usecase on a C based SoC performance model and collected statistics for every 10 microsecond time window at the initiator port. The stats captured includes read/write average latency, bandwidth and outstanding requests for the initiator port under consideration. The exact format of the data collected is shown in Fig 8. The choice for number of PCA components for normalized dataset came out to be 3 (using 80% variance criteria) and the number of clusters was set to 4 (using elbow method) as shown in Fig 9. These hyperparameters may differ depending upon the usecase, initiator port, time window size, performance metrics collected etc. Fig 10 shows the formed clusters along with their centroids. Each point in the figure essentially represents a time window (a row in our dataset). Our algorithm also reports the exact time windows which correspond to each cluster's centroid (denoted as trace snippets to be analyzed in RTL). Trace snippets are essentially the read/write address patterns initiated in that time window. So instead of running the complete 1 millisecond usecase trace in simulation, we run only the centroid traces(which adds to 40 microsecond for 4 clusters), significantly reducing the simulation time, while at the same time preserving the usecase properties.

	time_stamp	r_avg_outstanding	w_avg_outstanding	r_avg_latency	r_bandwidth	w_avg_latency	w_bandwidth
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	1.0	20.0	13.0	515.0	755.2	93.0	275.2
2	2.0	52.0	3.0	4995.0	627.2	429.0	192.0
3	3.0	54.0	3.0	2146.0	985.6	125.0	211.2
4	4.0	23.0	1.0	989.0	883.2	95.0	198.4
...	...	...	...	...	...	...	...

Fig 8: Sample input format to the k-means algorithm

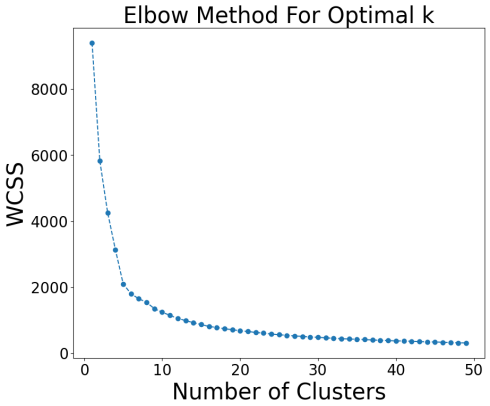


Fig 9: Determining optimal number of clusters

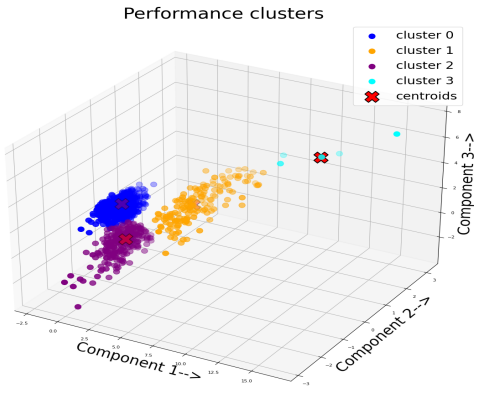


Fig 10: Visualization of clusters formed

Performance Simulations Failure Analysis (Big and Unique Buckets)

For each of our RTL performance tests we collect a large number of statistics, some of them are mentioned in the below table Fig 11. These stats are in addition to the performance metrics of the initiator port as described in the previous section. For each test, we are more interested in analyzing the correlation of the performance metrics (latency, bandwidth and max outstanding requests) with either the previous runs, or with the architecturally defined numbers (rather than their absolute numbers) . So performance stats are fed into the model as correlation coefficients (Fig 12.2). Additionally, for non-numeric features (e.g., Fabric Name), we employ suitable numeric mappings, a process refined through a series of experimental iterations to suit our test suites. Fig 12.1 represents the format in which data for each test is captured. It is subsequently normalized and the number of PCA components and clusters are determined using the approach described earlier.

DDR Activate count	***write_average_outstanding (wr_avg_os)***	***IP Max Outstanding (IP MO)***	***Fixed AxID (sameaid)***	***Address pattern (Linear/Random)***
DDR Refresh count	***Memory Management Unit Enable/Disable (MMU Enable)***	***Fabric Max Outstanding (FAB MO)***	***Cache hit enable/disable (SLC Hit)***	***Virtual Channel targeted (FAB VC)***
DDR Precharge count	***Memory Controller Latency***	***Memory Controller bandwidth***	***Memory controller bandwidth efficiency***	***Traffic Pattern (rw_pattern)***

Fig 11: Test Statistics

Testname	Portname	rw_pattern	IP MO	FAB MO	Fabric	Linear/Random	MMU Enable	sameaid	SLC Hit	...	write_bw
Test 0	Port 0	rw	128.0	128	Fabric 0	linear	mmudis	uniqueaid	slcmiss	...	1.000000
Test 1	Port 1	wo	48.0	48	Fabric 1	linear	mmudis	uniqueaid	slcmiss	...	0.004155
Test 2	Port 2	wo	64.0	64	Fabric 2	linear	mmudis	uniqueaid	slcmiss	...	0.001623
Test 3	Port 3	ro	32.0	32	Fabric 3	linear	mmudis	uniqueaid	slcmiss	...	0.000000
Test 4	Port 4	ro	64.0	64	Fabric 4	linear	mmudis	sameaid	slcmiss	...	1.000000
...	...	...	...	...	...	...	...	...	...	...	...

Fig 12.1: Sample input format to the k-means algorithm

Observed Latency (ns)	Expected Latency (ns)	Correlation
802	977	0.820880
879	981	0.896024
792	969	0.817337
714	969	0.736842
806	982	0.820774
771	994	0.775654

Fig 12.2: Performance metrics are fed in as correlation coefficients

Using 80% variance criteria, the number of PCA components comes out to be 5 which is significantly less than our total number of features(35). To ensure clustering efficiency while reducing the number of

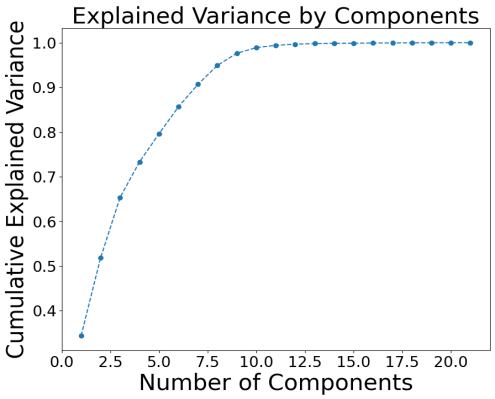


Fig 13: Determining optimum number of PCA components

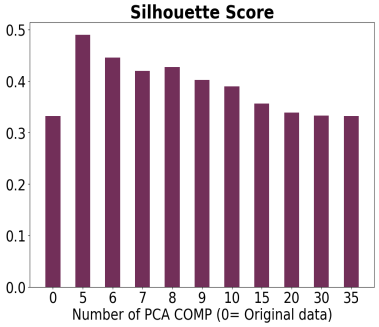


Fig 14: Silhouette Score vs PCA Components (Bar corresponding to 0 components is score on clustering with original 35 components)

components, we also plot the Silhouette score for different PCA components for a given number of clusters as shown in Fig 14.

Once hyperparameters are finalized, all identified failures are input into our algorithm, which generates clusters, each characterized by a centroid representing what we identify as priority debugs. Hence we begin with debugging the tests corresponding to centroids and notably witness the resolution of performance bottlenecks in other cluster members as well (attributed to the similarity of issues they share). This significantly improves our debug speed reducing it from an estimated 2 weeks to as low as 3 days. Fig 16 contains a visual representation of these clusters and their centroids, where each point essentially represents a single failing test. It is important to observe that we have used only the top 3 PCA components for plotting, though their total number is 5 as determined in Fig 15 (elbow method).

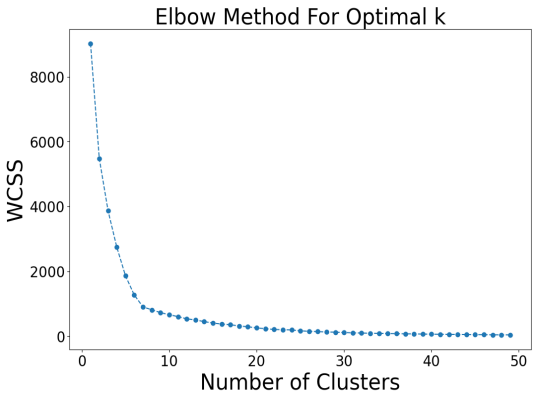


Fig 15: Determining optimal number of clusters

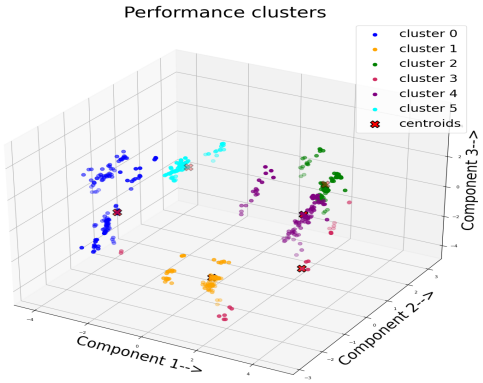


Fig 16: Visualization of clusters formed

In conjunction with this, our objective extends to the analysis of various cluster properties, to better understand the cause of failures. Various per cluster stats(for e.g mode of each feature for all points within a cluster), along with the feature values for the cluster centroids are captured in Fig 17. A large number of tests in a cluster, of a particular address pattern(or any other feature for instance) indicates the cause of failure for those cluster members. A high mode count for a feature corresponds to its low variance in the cluster. This kind of analytical approach(by observing variances) provides valuable insights into performance bottlenecks of that cluster's members.

Features	Variance Cluster 0	Mode Cluster 0	Mode count(%) Cluster 0	Centroid Cluster0	Variance Cluster 1	Mode Cluster 1	Mode count(%) Cluster 1	Centroid Cluster1
rw_pattern	0.000000	rw	100.000000	rw	0.000000	rw	100.000000	rw
IP MO	9532.929577	160.0	26.388889	160.0	12159.739229	256.0	16.161616	256.0
FAB MO	9532.929577	160	26.388889	160	12159.739229	256	16.161616	256
Fabric	0.000000	Fabric 6	100.000000	Fabric 6	0.000000	Fabric 2	100.000000	Fabric 2
Linear/Random	1588.419405	linear	80.555556	linear	1076.066790	linear	87.878788	linear
MMU Enable	0.000000	mmudis	100.000000	mmudis	0.000000	mmudis	100.000000	mmudis
sameaid	0.000000	uniqueaid	100.000000	uniqueaid	391.671820	uniqueaid	95.959596	uniqueaid
SLC Hit	890.062598	slcmiss	90.277778	slcmiss	1436.817151	slcmiss	82.828283	slcmiss
Structural Latency	17.996870	154.0	37.500000	155.0	20.286539	192.0	51.515152	192.0
FAB VC	357902.973396	Type 0	72.222222	Type 0	13636.363636	Type 3	54.545455	Type 3

Fig 17: Per cluster statistics

VI Conclusion

We proposed an efficient AI-Based algorithm for stimulus generation and performance failure analysis. This stimulus generation technique helped cut down the simulation time by 96% (from 120 to 6 hours) with multiple unique snippets running simultaneously to achieve the same full use-case behavior. AI-Based Performance Analysis technique helped identify the outlier Subsystem, IP port and Test Scenario to prioritize the debug. Using this data, we identified bugs related to MMU Enable, IP specific credit/buffer sizes early in the design cycle that got fixed in the same project.

This methodical and highly optimized approach to performance verification flow helped detection of performance gaps early in the design cycle. It helped uncover multiple bugs leading to RTL fixes in time for achieving critical project milestones.

One of the many RTL issues we found during our experiments was unintentional clock changes in one particular Fabric configuration. Our algorithm generated a significant cluster, encompassing all latency tests from all routes involving initiators from that exact same fabric. Upon closer examination of the cluster's centroid, we could swiftly identify the clock frequency issue which led to resolving multiple failures.

We aim to further enhance our existing flow by incorporating additional input test statistics to enhance the accuracy of the results of our algorithm. Also, we intend to augment our current simulations transitioning from cold cache state to dumping all caches/buffers states and initialize them in RTL prior to running the snippets. Furthermore, we plan to extend our experiments to include running test scenarios with multiple initiators and by using different time windows.

References

1. S. Song, R. Desikan, M. Barakat, S. Sundaram, A. Gerstlauer and L. K. John, "Fine-Grain Program Snippets Generator for Mobile Core Design", GLSVLSI '17: Proceedings of the on Great Lakes Symposium on VLSI, 2017
2. G Anthony; Ronald G. Dreslinski; Thomas F. "Full-system analysis and characterization of interactive smartphone applications" IEEE International Symposium on Workload Characterization, 2011.
3. Z. Poulos and A. Veneris, "Clustering-based failure triage for rtl regression debugging", *IEEE Int'l Test Conference*, 2014.
4. A. Truong, D. Hellström, H. Duque, L. Viklund, "Clustering and Classification of UVM Test Failures Using Machine Learning Techniques" , DVCON Europe 2018
5. Khaled A. Ismail, Mohamed A. Abd El Ghany "Survey on MachineLearning Algorithms Enhancing the Functional Verification Process" *Electronics*, *10*(21), 2688, 2021
6. El Mandouh, E.; Wassal, A.G., "Accelerating the debugging of FV traces using K-means clustering techniques". In Proceedings of the 11th International Design & Test Symposium (IDT), Hammamet, Tunisia, 18–20 December 2016; pp. 278–283, 2016
7. S. Sokorac, Optimizing random test constraints using machine learning algorithms. In Proceedings of the Design and Verification Conference (DVCON), San Jose, CA, USA, 27 February–2 March 2017
8. K. R. Shahapure and C. Nicholas, "Cluster Quality Analysis Using Silhouette Score," 2020 IEEE 7th International Conference on Data Science and Advanced Analytics (DSAA), pp. 747-748 Sydney, NSW, Australia, doi: 10.1109/DSAA49011.2020.00096, 2020