A Topological Approach to Hardware Bug Triage

Rico Angell, Ben Oztalay and Andrew DeOrio
Dept. of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor
{rangell, boztalay, awdeorio}@umich.edu

***Abstract*—Verification is a critical bottleneck in the time to market of a new digital design. As complexity continues to increase, post-silicon validation shoulders an increasing share of the verification/validation effort. Post-silicon validation is burdened by large volumes of test failures, and is further complicated by root cause bugs that manifest in multiple test failures. At present, these failures are prioritized and assigned to validation engineers in an ad-hoc fashion. When multiple failures caused by the same root cause bug are debugged by multiple engineers at the same time, scarce, time-critical engineering resources are wasted.**

Our scalable bug triage technique begins with a database of test failures. It extracts defining features from the failure reports, using a novel, topology-aware approach based on graph partitioning. It then leverages unsupervised machine learning to extract the structure of the failures, identifying groups of failures that are likely to be the result of a common root cause. With our technique, related failures can be debugged as a group, rather than individually. Additionally, we propose a metric for measuring verification efficiency as a result of bug triage called Unique Debugging Instances (UDI). We evaluated our approach on the industrial-size OpenSPARC T2 design with a set of injected bugs, and found that our approach increased average verification efficiency by 243%, with a confidence interval of 99%.

I. Introduction

Increasing transistor counts continue to drive the increasing complexity of digital designs. As a result, the verification burden has increased dramatically, to the point where the number of verification engineers nearly equals the number of design engineers, on average [1]. The efficient use of verification engineering resources has a critical impact on a product's time to market. In particular, effort spent during post-silicon validation triaging, diagnosing, and debugging failures that slip through pre-silicon verification is currently a manual, time-consuming process. A critical inefficiency occurs when multiple verification engineers spend days debugging multiple failure reports, only to reach the same root cause bug.

Post-silicon validation begins when the first prototype chips become available. In contrast to slow, limited scale pre-silicon verification, post-silicon prototypes have the advantage of executing the entire design at full speed. High execution speeds enable high coverage, and post-silicon platforms run additional directed and pseudo-random tests. As a result, post-silicon validation produces many test failures. A test failure usually generates a failure report with some information about signal activity within the design. Failure reports are stored in a database, and validation engineers inspect the accumulating reports, manually triaging them and assigning the failures to the engineering team for debugging.

Ideally, each engineer on the validation team would debug a test failure and each would locate a different root cause bug. Unfortunately, a root cause bug may result in more than one test failure, and multiple engineers may not know they are working on the same problem until they have spent valuable time (often days) debugging. This frustration comes when schedule pressure is high, and resources are limited. Debugging on real hardware is further complicated by non-deterministic executions. Environmental variations such temperature cause subtle timing perturbations that result in different test outcomes, even for multiple executions of the same test.

Multiple engineers may debug different test failures related to the same root cause bug as a result of present ad-hoc bug triage methods. Not only is current bug triage inaccurate, it can also be a time-consuming and manual process. Furthermore, it only groups test failures by the development team that is best suited to handle them. Once test failure reports have made it to the team level, they are assigned to engineers in an ad-hoc fashion, where related test failures may be unknowingly distributed to multiple engineers.

A. Contributions

Our goal is to increase the efficiency of verification engineers' time through automated, accurate hardware bug triage. Our approach starts with a database of test failure reports, and automatically groups the reports, where reports within a group are likely to share a common root cause. We extract features of the failure reports using a graph-based model representing the underlying hardware design's topology. This provides the input to a machine learning algorithm, which clusters the failure reports. Verification engineers can now debug failure reports as a group rather than individually, avoiding two engineers working on the same root cause bug. To measure the impact of our techniues on the efficiency of post-silicon validation, we propose a new metric called *Unique Debugging Instances (UDI)*, which measures duplication of effort. Our system improves the verification process by:

- Using topology-aware feature extraction and machine learning
- Tolerating non-deterministic test executions
- Decreasing the frustrating situation where multiple engineers arrive at the same root cause bug
- Increasing efficiency of verification engineering resources
- Introducing the Unique Debugging Instances (UDI) metric to quantify verification debugging efficiency

II. Topology-aware hardware bug triage algorithm

Our topology-aware hardware bug triage technique begins with a database of reports from tests that fail during post-silicon validation. These failure reports are combined with topological information about the hardware design, which is generated in parallel with the test failure reports. We use the results of both of these steps to create clusters of test failure reports that share a common root cause. Clusters of failure reports can then be sent to engineers for fast, efficient debugging. Figure 1 shows an overview of the process.

The test failure reports contain information about the test being run and when it failed, along with information about the state of the hardware at the time of the failure, such as the values being carried by a subset of the design's signals around the time of failure.

While the test failure reports are gathered, we perform a static analysis of the design under test to build a directed graph representing the hardware topology. In this graph, vertices are gate level pieces of logic, while the edges are signals connecting the logic.

Our system then partitions the graph representation of the hardware into a number of subgraphs. Next, we process the resulting group of subgraphs to build groups of logically related signals. Finally, the groups of related signals are filtered according to whether or not they contain at least one of the signals included in the failure reports, and those without any are ignored.

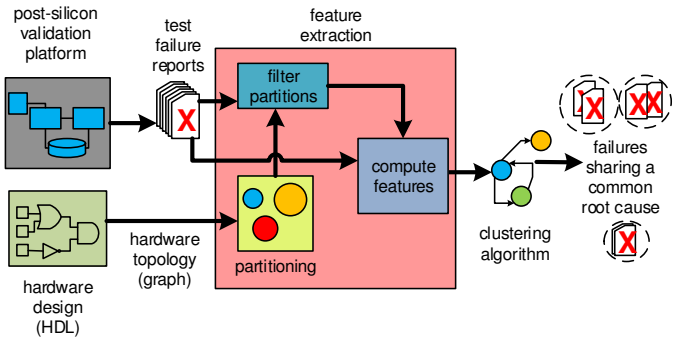

Fig. 1. **Our hardware bug triage algorithm** takes as input a database of test failure reports and the design's HDL. Using the HDL, we create topology-aware subgraphs of the design. Then, features for a machine learning algorithm are extracted, considering both the subgraphs and failure reports. These features are used to cluster the failure reports, where failure report in the same cluster share a common root cause.

Lastly, the failure reports are read to find the values of each of the signals in the signal groups, which are used to compute features that represent each respective test failure to an unsupervised machine learning algorithm. The final output of this machine learning step is a set of clusters of failure reports, where the test failures are clustered by their common root causes of failure.

These groups of related test failures can then be passed on to engineers for debugging, where the clustered test failures give each engineer more information to debug with, while reducing the number of engineers that are simultaneously working toward finding and fixing the same causes of failure.

A. *Extracting the Hardware Topology*

Hardware bugs can affect neighboring hardware that is closely connected to the root cause components. We extract these connectivity relationships by examining a design's topology.

Beginning with a hardware description language (HDL) model, we build a graph representing the design. For example, the OpenSPARC T2, which we use later in our experiments, is written in Verilog HDL. In this directed graph, logic, primary inputs and primary outputs are vertices, and the signals that connect logic, inputs and outputs are edges. Figure 2 shows an example of the transformation from logic connected by signals to vertices connected by edges.

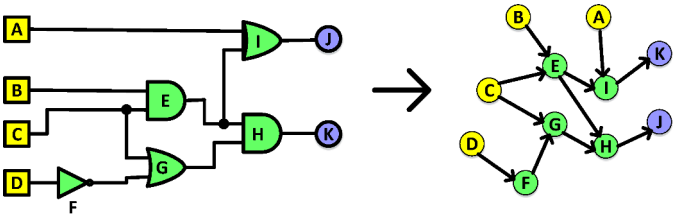

Fig. 2. **Converting the hardware design to a graph:** We construct a representation of the logic design by means of a graph such that each vertex corresponds to a piece of logic and each edge corresponds to a wire. This process lets us extract information about the hardware design's topology, in order to better inform our machine learning algorithm.

Next, we partition the graph into subgraphs, representing distinct sections of logic. The goal of the partitioning algorithm is to cut as few edges as possible for the specified number of subgraphs. Thus, each subgraph is a collection of closely connected hardware components. Figure 3 shows the relation between sections of the hardware design and their corresponding subgraphs.

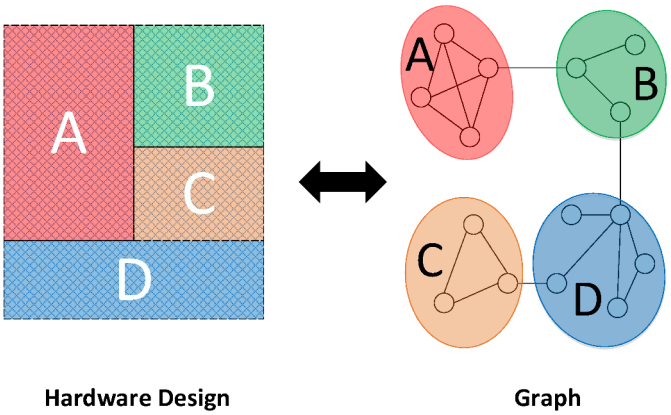

Fig. 3. **Partitions of the graph and the hardware design,** Edges, and their corresponding signals, are deterministically assigned to groups after the vertices are partitioned. The edge groups are used with information about the signal values at the time of failure to calculate features for our machine learning algorithm.

We employed the popular and very fast graph partitioning library ParMETIS to partition the graph of the OpenSPARC T2, which has over 3 million vertices. ParMETIS is a parallel version of METIS, which progressively coarsens a graph by removing vertices, then partitions the coarsened graph. Then, it refines the partition while the iteratively uncoarsening the graph by adding vertices back, until the graph is its original size and the partitioning is finalized [2].

The number of subgraphs is flexible, though it is chosen with the goal of achieving a balance of granularity in the subgraphs and information about how signals in the design are topologically related.

Next, our algorithm processes the subgraphs to produce a group of signals associated with each subgraph. This is done by collecting all of the outgoing edges of the vertices in each subgraph. The end result is a set of groups of topologically related signals. The groups without any of the signals included in the failure reports are discarded, and those remaining are used to construct features for each failure report in the machine learning step.

B. *Computing Features*

Feature selection is critical to accurately clustering, and thus effectively triaging, the failure reports. The features should have the right level of detail, be relatively simple to compute, be proportional to bug activity, and describe a wide range of design activity. Satisfying all of these properties is a difficult problem, and our goal is to balance these priorities. We approach the problem of satisfying these four properties by selecting features related to the design's topology.

To compute the features that will represent each failure report to the machine learning algorithm, we must reduce the raw partitions described in the previous section to filtered partitions that contain only the observable signals in the design, while maintaining the integrity of the partitions with respect to the topology. The observable signals are the ones whose values are included in the failure reports. After filtering the partitions for observable signals, we iterate over the failure reports and each partition. For each partition, we average over the signal values (0 or 1) of the wires in the partition. The result is a feature vector whose dimension is the number of filtered partitions, meaning that one feature vector represents a single failure report. These feature vectors are used by the machine learning algorithm to cluster the failure reports.

More specifically, let $\mathcal{U}$ be the set of observable signals in the test failures. Let $F = \{f_1, f_2, \ldots, f_m\}$ be the set of failure reports that

include signal values for each $s \in \mathcal{U}$. Let $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ be a partitioning of $\mathcal{U}$ (i.e. $\bigcup_{i=1}^{n} P_i = \mathcal{U}$ and for each $P_i, P_j \in \mathcal{P}, i \neq j, P_i \cap P_j = \emptyset$). Define the mapping ϕ as follows

$$\phi : F \times \mathcal{U} \rightarrow [0, 1].$$

Essentially, ϕ computes the value of a single feature from a particular test failure. Let $X = \{\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_m\}$ be the set of feature vectors that correspond to the test failures reports in F. Each vector in X is defined as

$$\vec{x}_i = [x_{i,1}\ x_{i,2}\ \cdots\ x_{i,n}]$$

where

$$x_{i,j} = \sum_{w \in P_j} \frac{\phi(f_i, w)}{|P_j|}.$$

Each vector, $\vec{x}_i$, is intended to have a relatively small Euclidean distance to other vectors that represent failures caused by the same bug and a relatively large Euclidean distance to vectors that represent failures caused by different bugs. The accuracy of this representation is critical to correctly grouping the failure reports in the machine learning phase.

C. Machine Learning

The features computed from partitions of signals produced in the partitioning step represent each test failure, and form the input to a machine learning algorithm. The machine learning algorithm produces a clustering of the test failures, where the test failures within each cluster are likely to be the result of the same root cause bug.

In our application of machine learning, the goal is to group test failures that are related by the root cause bug behind the failure, without any prior knowledge of the root cause. Additionally, the number of bugs that were exposed by the tests is unknown. These two factors led us to apply an *unsupervised* machine learning algorithm. Unsupervised, unlike *supervised*, machine learning algorithms do not required labeled training data. *Labels* are names given to the clusters produced by a machine learning algorithm. In our case, the labels would be the actual bugs that caused the tests in a particular cluster to fail. Our technique does not require any such prior knowledge.

The particular machine learning algorithm we chose was k-means clustering, because it is simple, fast, and proven to be scalable. In addition to feature vectors, k-means requires the number of clusters as an input parameter. Thus, the performance of our approach relies, in part, on choosing a good number of clusters. Ideally, the chosen number of clusters would match the actual number of bugs in the hardware. This would result in the most homogeneity within the clusters, where the ideal outcome is complete homogeneity. While the number of bugs cannot be predicted outright, it can be estimated using an approach presented in [3], [4]. We further explore the impact of the number of clusters in our experimental results.

Once the test failures have been clustered, the clusters can be given to engineers, who can debug the test failures as a related group. This provides the engineers with more information about each bug while reducing the number of engineers that are simultaneously working toward finding and fixing the same bug.

The computational complexity of our technique lies in ParMETIS and *k*-means. Both algorithms solve problems that are known to be NP-hard. However, both ParMETIS and *k*-means are well established heuristic algorithms, and been proven scalable with industrial-size inputs. In practice, we found them to be effective on the industrial-size OpenSPARC T2 design.

III. Experimental Evaluation

We evaluated our approach on the OpenSPARC T2 design [5] running test workloads with a set of injected bugs. We simulated the noisy post-silicon environment by varying the random seeds, which alters communication timing and traffic. The database of failure reports used in our evaluation was the result of many simulations of these failures, testcases and injected noise. In our evaluation, we explore how the number of clusters, partitions of the graph (number and granularity of features), and unique bugs in the bank of failures affect the accuracy of our algorithm.

We used two metrics throughout our evaluation. First, we used Normalized Mutual Information (NMI) to compare the effectiveness of multiple executions of our algorithm against one another. NMI is a common metric used in machine learning to evaluate how well a clustering corresponds to a known correct clustering *(ground truth)*. Specifically, NMI measures the mutual dependence of two sets. The best case is total dependence between the ground truth and clusters produced by the algorithm, which is represented by an NMI of 1.0. The worst case is no dependence between the ground truth and the clusters given from the algorithm, which is represented by an NMI of 0.0. Throughout our experiments, the ground truth is the grouping of the failure reports by the known location of each bug. Note that the known bug locations are used only for evaluation, since our technique requires no prior knowledge of the bugs.

In addition to the traditional machine learning metric of NMI, we also evaluated the effectiveness of our technique in decreasing duplicate verification engineering effort. We quantify the impact of our algorithm on verification efficiency with a metric we call Unique Debugging Instances (UDI). A low value indicates little duplicate effort, while a high value indicates more duplicated effort by the verification engineering team. We explain UDI in depth in Section III-F. Then, in our last experiment, we compare our method against a baseline bug triage approach.

A. Experimental Setup

We used the OpenSPARC T2 design [5] for evaluation. In each simulation, a root cause bug is injected into the design at cycle 10,000. At cycle 10,100, the values of the control signals in 10 modules are recorded in a failure report. In order to generate our database of failures, we simulated each of the testcases found in Table I, with each of 10 random seeds, and each of 50 bugs that evenly span those 10 modules in the design (5 bugs in each module). The modules are described in Table II. Thus, we had a total of 5,000 test executions. Of these 5,000 test executions, 3,750 resulted in failure reports. The failed simulations comprise our database of failure reports used in our experiments.

Testcase	Description	Length (cycles)
blimp_rand	hypervisor test	251,480
fp_addsub	floating point add/subt	913,093
fp_muldiv	floating point mult/div	238,343
isa2_basic	constrained-random	452,009
isa3_asr_pr	constrained-random	1,178,151
isa3_window	constrained-random	1,282,348
mpgen_smc	constrained-random	135,251
ldst_sync	thread sync. instrs.	64,570
n2_lsu_asi	load/store unit test	62,523
tlu_rand	trap logic unit test	591,434

TABLE I
Workloads simulated on OpenSPARC T2, which are part of the regression suite provided with the design's distribution.

Module	Description	Control Signals
EXU	execution unit	265
DEC	decoder	36
FGU	floating point and graphics unit	197
GKT	gasket interface	71
IFU	instruction fetch unit	270
MMU	memory management unit	420
PKU	thread pick unit	509
PMU	performance monitoring unit	156
LSU	load/store unit	533
TLU	trap logic unit	452

TABLE II

OpenSPARC T2 units injected with bugs. We evenly distributed 50 bugs among these 10 modules, with 5 in each.

B. Partitioning OpenSPARC T2

As part of our algorithm's feature extraction step, we extracted the topology of the design and created partitions which are used later by the machine learning step. The OpenSPARC T2 design has approximately 3 million gates. We compiled the design, which generates a netlist as an intermediate step. The netlist is then reduced to a graph using the methodology presented in Section II-A. After converting the OpenSPARC T2 design into a graph, we partitioned the graph into 100, 500, 1,000, 5,000, and 10,000 partitions.

During post-silicon validation, only a small subset of the design's signals are accessible for observation. To model this, we record the value of 2,909 control signals during each simulation. Since not all of the nearly 3 million gates and their corresponding wire outputs in the design are observable in our log files, we filter the partitions for the observable signals. We filtered the partitions by removing any unobservable wires, but keeping the same structure of the partitions provided by the partitioning algorithm. These "raw" partitions returned by the partitioning algorithm include both observable and unobservable signals. If a raw partition is completely composed of unobservable signals, it is discarded. We provide more detail about the different partitionings and the effect they have on our algorithm in Section III-C.

Raw Partitions	Average Size	Median Size	Std Dev
100	27,858	25,465	6,959
500	5,572	5,009	1,742
1,000	2,786	2,463	1,048
5,000	557	468	301
10,000	279	222	185
Filtered Partitions	**Average Size**	**Median Size**	**Std Dev**
80	52	19	104
264	16	6	41
381	11	4	25
817	5	2	9
1,057	4	2	6

TABLE III

Number of partitions before and after filtering, with sizes measured in number of signals. We filter the raw partitions for signals that are observable during simulated post-silicon validation. The filtered partitions form the input to our algorithm.

After producing the database of test failure reports and partitioning the design, we tested the machine learning step of our algorithm. In the next sections, we explore the effect that changing the parameters of our system (number of partitioning, number of clusters, and number of bugs) had on the performance of the machine learning step.

C. Effect of Number of Partitions

Our first experiment focused on the impact of the number and granularity of features on the quality of our test failure clusterings. We executed our algorithm for different partitionings of the OpenSPARC T2 design, shown in Table III, using all of our failure reports and clustering to 50 clusters. The number of features used by the algorithm is exactly the number of filtered partitions, which are those that include only observable signals, for each partitioning of the design (e.g. for 5,000 raw partitions, our algorithm extracts 817 features from each of the failure reports for clustering). We measured the NMI of the clustering after each execution of our algorithm and plotted the data in Figure 4. We observed that the highest granularity and number of features resulted in the highest NMI, but the difference between the partitionings was well within the error of the k-means. Hence, the difference we saw between partitioning schemes is minimal, but we chose to use the 10,000 raw partitioning for its granularity. Additionally, the number of partitions had no significant impact on the overall run time of our system, as the time to partition the OpenSPARC T2 design was under 1 minute.

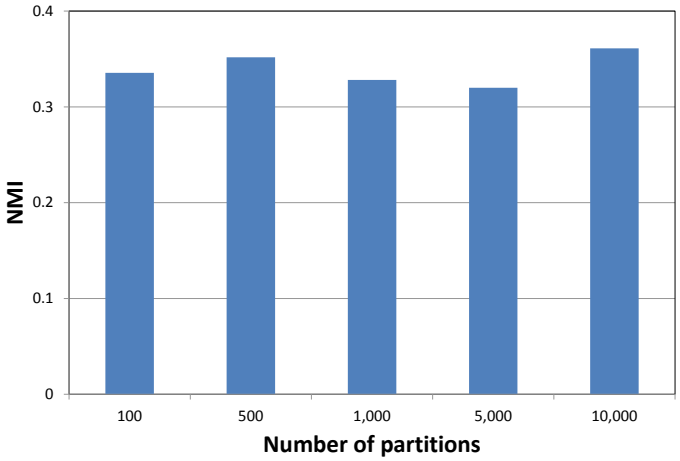

Fig. 4. **Effect of the number of partitions** on the quality of clusters. We observed that the quality of clusters was close with different numbers of partitions, and was maximized with 10,000 partitions. Intuitively, higher granularity yielded better results, since more partitions leads to more features as input to our algorithm. This result led us to choose 10,000 raw partitions as our default partitioning.

D. Effect of Number of Clusters

Our next experiment assesses how our algorithm responds to different numbers of clusters provided as a parameter to k-means clustering. We executed our algorithm for numbers of clusters ranging from 2 through 50. Next, we compared the resulting clusters to an ideal clustering, where clusters contained only failures from the same root cause. We measured NMI for each execution of the algorithm and plotted the data in Figure 5. When we clustered with very low numbers of clusters, we observed several outliers compared to the overall trend. These outliers can be resolved by the bug forecasting methods mentioned in Section II-C. Overall, we observed a trend where the highest NMI was where the number of clusters was equal to the number of bugs.

E. Effect of Number of Bugs

Our algorithm is affected by both the number of root cause bugs and the number of clusters. We ran our algorithm on every combination of number of bugs and clusters and recorded the NMI of the resulting clusters of test failures. The results are displayed as a heatmap in Figure 6, with the number of clusters on the X-axis and number of injected bugs on the Y-axis. Overall, we observed a trend where the highest NMIs were rightward and upward in the heatmap, corresponding to when the number of bugs was equal to

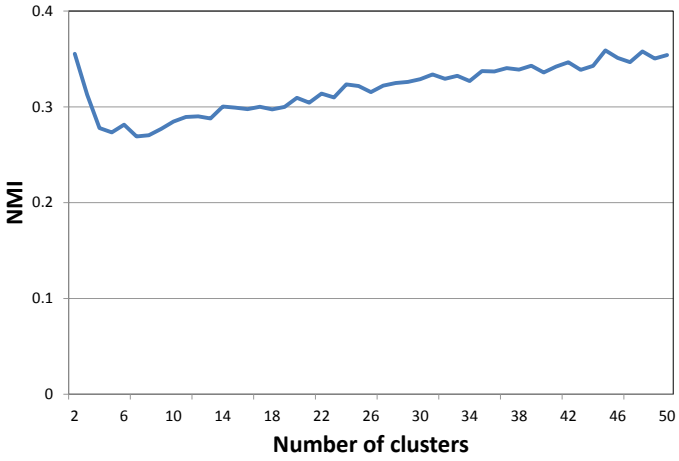

Fig. 5. **Effect of number of clusters** on the quality of clusters, expressed as NMI. This experiment used test failures resulting from 50 injected bugs and used 10,000 raw partitions. The quality of the clusters followed a trend of increasing as the number of clusters approached the number of bugs.

the number of clusters. However, the NMI was consistently better for each number of bugs with more clusters. The deviation from a clear diagonal trend and the hotspot around 25 bugs and 50 clusters is due to the similarity of several bugs to each other, for example, bugs in injected in the same module.

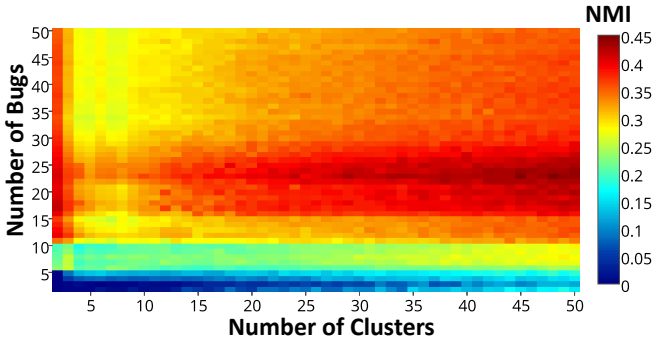

Fig. 6. **Number of bugs vs. number of clusters.** Each point in this heat map represents one execution of our algorithm with 10,000 partitions. We varied both the number of clusters (X-axis) and the number of root cause bugs present in the database of failure reports (Y-axis). The NMI of the resulting clustering is plotted as a color (Z-axis). While following a general rightward and upward trend, the hotspot around 25 bugs and 50 clusters is due to the similarity of several bugs to each other, for example, bugs in injected in the same module.

F. Impact on Verification Efficiency

We also evaluated the impact of our bug triage technique on the efficiency of the verification process. We note that time to market is directly impacted by the efficient use of post-silicon validation engineering resources. At present, post-silicon failure reports are triaged largely in an ad-hoc fashion. When a single root cause bug results in multiple test failures, it may be debugged by multiple engineers simultaneously, wasting valuable engineering resources. Reducing the amount of duplicate debugging effort is the goal of our system.

Engineers can debug similar test failures more quickly than very different failures. Hence, the best-case scenario would be where there is only one root cause bug in the group of failure reports assigned to an engineer, and no two engineers would see failures caused by the same bug. On the other hand, the worst-case scenario would be if every engineer saw a failure report caused by each root cause bug. Then, every engineer would debug every bug separately, maximizing the duplicated debugging effort. This is the motivation behind our metric which we call Unique Debugging Instances (UDI).

UDI aims to represent the total amount of significant debugging effort of a team of engineers during the debugging process. To accomplish this, the metric counts the number of unique root causes in each group of failure reports assigned to different engineers and sums over all the groups. UDI can be applied universally, but is affected by several factors beyond the quality of failure report clusters, including the number of engineers and the number of failure reports.

To see how verification effort is improved by our technique, we use UDI to measure a baseline approach resembling the ad-hoc bug triage in modern designs. We model this with a uniform random distribution of failure reports among 50 clusters. Since each distribution will be different, we use a Monte-Carlo method. We ran both our algorithm, with 50 clusters and 10,000 raw partitions as input parameters, and the bug distribution 100 times. For both solutions, we measured the UDI of the groups produced by each iteration. Figure 7 displays a histogram representing the results. The blue (dark) bars are the results from the executions of our algorithm while the red (light) bars are the results from the Monte Carlo approach of assigning failure reports to engineers. We observed a mean of 782 UDI and a standard deviation of 17.4 UDI for our algorithm and a mean of 1,903 UDI and a standard deviation of 15.9 UDI for the Monte Carlo approach. Thus, our algorithm resulted in less duplicated engineering effort, compared with the baseline. This yields an increase in average verification efficiency of 243%, with a confidence interval of 99%.

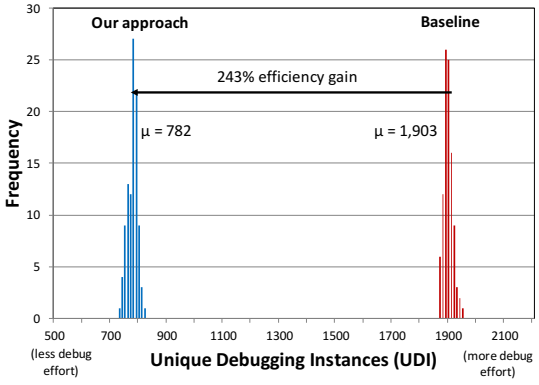

Fig. 7. **Impact on verification efficiency,** measured in Unique Debugging Instances (UDI) using a Monte-Carlo method. Smaller UDI indicates less duplication of engineering effort. The baseline algorithm makes random assignments of failure reports to clusters. The figure displays the distribution of UDI of 100 executions of our algorithm run with 50 clusters and 10,000 raw partitions and the UDI of 100 executions of a uniform random grouping over the same data set. Our approach resulted in a mean of 782 UDI with a standard deviation of 17.4 UDI, while the Monte Carlo approach resulted in a mean of 1903 UDI and a standard deviation of 15.9 UDI. Thus, our result increases average verification efficiency by 243% with a confidence interval of 99%.

IV. Limitations

While our technique is effective in increasing post-silicon validation efficiency, it has some limitations. Our algorithm performs best when root cause bugs result in multiple failure reports. On the other hand, it is less effective with smaller quantities of data, for example when each failure report is the result of only one root cause bug.

Similar to other applications of machine learning, our application is limited by the quantity of data available. This includes both the number of failure reports as well as the number and quality of signals whose activity is recorded along with a test failure.

Additionally, estimating the number of bugs affects the performance of our algorithm due to the necessary input of k-means clustering. This can be alleviated in part by methods of estimating the number of bugs present in a hardware design [3], [4].

V. Related Work

Unsupervised machine learning has been applied to many problems, such as grouping related news articles and other documents [6]. Another application of unsupervised machine learning is finding communities within social networks [7]. Specifically, all of these applications share in common that their input data consists of unlabeled examples. Their goal, as is the goal of hardware bug triage, is to group similar, unlabeled pieces of information.

The method of modeling problem spaces as graphs and partitioning those graphs has been applied to a wide range of problems in software and hardware. Researchers developing a hypergraph partitioning algorithm have tested and tuned it on graphs of VLSI designs [8], which has many applications. In the software world, the authors of [9] developed a tool to automatically generate a graph representation of software systems, then partition that graph to find the system's submodules and make it easier for a developer to understand a system. As presented in [10], graph partitioning can be used to find the best mapping of parallel computing tasks to a given hardware architecture to reduce communication overhead between dependent tasks. This technique has been applied to the problem of emulating networks in [11], which resulted in a significant performance boost when emulating networks. In contrast to these applications, our technique applies graph partitioning to a hardware design to compute powerful, descriptive features that later enable a machine learning algorithm to better understand the topology of the design.

Automated bug triage has been explored for software projects that must address large volumes of bug reports, as in [12] where machine learning was used to classify bug reports. The authors of [13] used supervised machine learning to automatically send bug reports to the appropriate development team. Supervised machine learning has also been used to predict which part of the source code caused a bug [14], and to identify duplicate bug reports [15]. The problem of triaging hardware bugs is different from that of software bugs in significant ways. While software bug reports are usually written by humans and contain a description of the problem, hardware test failure reports are typically automatically generated and may not contain much human-readable information. Information included in a hardware test failure report may be binary data describing signal activity and other system state at the time of failure. The effectiveness of hardware bug triage relies on extracting meaningful features from this binary data.

Another method of triaging hardware test failure reports is proposed by [16], which relies on assertions and SAT-based design debugging data to group similar test failures. The authors of [17] use data mining and an affinity-propagation clustering algorithm to produce improved test failure clusters while finding failures within each cluster that are representative of the rest of the group, providing a debugging priority ranking within clusters. While effective on small designs (2,000 to 83,000 gates), the scalability is limited by SAT's state space explosion problem. On the other hand, our system quickly triages failure reports from industrial-sized designs such as the OpenSPARC T2, which has approximately 3 million gates.

VI. Conclusions

We have presented a technique for hardware bug triage that leverages a design's topology to extract powerful descriptive features from post-silicon test failure reports. These features are computed using graph partitioning on a graph representation of the hardware design. When provided as input to an unsupervised machine learning algorithm, the features enable accurate grouping of test failure reports, which can then be debugged as a group. This enables more efficient use of verification engineering resources.

In our experimental results, we deployed our technique on the industrial-size OpenSPARC T2 design with a set of injected bugs and a database of test failure reports. When converted to a graph, the design had millions of edges and vertices. We use a new metric, Unique Debugging Instances (UDI) to measure the duplication of effort. With this metric, we found that average verification efficiency increases by 243%, with a confidence interval of 99%. Our approach reduces duplication of engineering effort during post-silicon debugging, helping to decrease a new product's time to market.

References

[1] Wilson Research Group, "Wilson research group functional verification study," 2012.

[2] G. Karypis and K. Schloegel, "ParMETIS: Parallel graph partitioning and sparse matrix ordering library," 2013.

[3] Q. Guo, T. Chen, Y. Chen, R. Wang, H. Chen, W. Hu, and G. Chen, "Pre-silicon bug forecast," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 3.

[4] Z. Poulos, Y.-S. Yang, and A. Veneris, "A failure triage engine based on error trace signature extraction," in *Proc. On-Line Testing Symposium*, 2013.

[5] "Sun microsystems OpenSPARC," http://opensparc.net/.

[6] X. Hu, X. Zhang, C. Lu, E. K. Park, and X. Zhou, "Exploiting wikipedia as external knowledge for document clustering," in *Proc. SIGKDD*, 2009.

[7] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 3-5, pp. 75 – 174, 2010.

[8] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: Application in VLSI domain," *IEEE Trans. VLSI Systems*, vol. 7, no. 1, pp. 69 – 79, 1999.

[9] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner, "Bunch: a clustering tool for the recovery and maintenance of software system structures," in *Proc. IEEE International Conference on Software Maintenance*, 1999.

[10] B. Hendrickson and R. Leland, "An improved spectral graph partitioning algorithm for mapping parallel computations," *SIAM Journal on Scientific Computing*, vol. 16, no. 2, pp. 452 – 469, 1995.

[11] K. Yocum, E. Eade, J. Degesys, D. Becker, J. Chase, and A. Vahdat, "Toward scaling network emulation using topology partitioning," *IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems*, pp. 242 – 245, 2003.

[12] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," in *Proc. ICSE*, 2003.

[13] D. Čubranić and G. Murphy, "Automatic bug triage using text categorization," in *Proc. SEKE*, 2004.

[14] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Trans. Software Engineering*, vol. 31, no. 6, 2005.

[15] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proc. ICSE*, 2008.

[16] Z. Poulos and A. Veneris, "Clustering-based failure triage for RTL regression debugging," in *Proc. ITC*, 2014.

[17] Z. Poulos and A. Veneris, "Exemplar-based failure triage for regression design debugging," in *Proc. LATS*, 2015.