Machine Learning-based Anomaly Detection for Post-silicon Bug Diagnosis

Andrew DeOrio, Qingkun Li, Matthew Burgess and Valeria Bertacco
Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI 48109
{awdeorio, qinkunl, mattburg, valeria}@umich.edu

ABSTRACT

The exponentially growing complexity of modern processors intensifies verification challenges. Traditional pre-silicon verification covers less and less of the design space, resulting in increasing post-silicon validation effort. A critical challenge is the manual debugging of intermittent failures on prototype chips, where multiple executions of a same test do not yield a consistent outcome.

We leverage the power of machine learning to support automatic diagnosis of these difficult, inconsistent bugs. During post-silicon validation, lightweight hardware logs a compact measurement of observed signal activity over multiple executions of a same test: some may pass, some may fail. Our novel algorithm applies anomaly detection techniques similar to those used to detect credit card fraud to identify the approximate cycle of a bug's occurrence and a set of candidate root-cause signals. Compared against other state-of-the-art solutions in this space, our new approach can locate the time of a bug's occurrence with nearly 4x better accuracy when applied to the complex OpenSPARC T2 design.

1. INTRODUCTION

The complexity of modern chips intensifies verification challenges, and as a result an increasing share of the verification effort is shouldered by post-silicon validation. Focusing on the first silicon prototypes, post-silicon validation poses critical new challenges such as *intermittent failures*, where multiple executions of a same test do not yield a consistent outcome. These are often due to on-chip asynchronous events and electrical effects, leading to extremely time-consuming, if not unachievable, bug diagnosis and debugging efforts.

Today, post-silicon validation is a largely manual, ad-hoc process. Beginning with the first silicon prototypes, chips are connected to a validation platform, which runs large volumes of tests at high speed. Test outputs are then checked against a reference model, or alternatively, they may self-check. When test outcomes match, testing progresses. However, when they do not match, a failure is identified and manual debugging begins. First the failure must be reproduced, a challenge in itself for bugs that are sensitive to subtle on-chip variations. When these bugs occur, different executions of the same test yield different results: some pass, while others fail. Often, it is the failing executions that are most difficult to obtain. Our focus is on these most difficult intermittent post-silicon bugs.

The goal of the human effort during post-silicon validation is to understand the correct operation of the design, and identify the root cause of any deviation from correct operation, that is, the cycle and critical signals involved in a bug occurrence. Machine learning, a branch of artificial intelligence, shares a similar goal, learning from examples and identifying the structure in a system. The large volume of data generated by many post-silicon test executions suggests that the application of machine learning is a promising solution, especially anomaly detection. For example, anomaly detection is often applied to automatically identify credit card fraud. In a similar manner, this approach might be applied to identify hardware bugs. Our goal is to unlock the potential of anomaly detection techniques in the context of post-silicon debugging.

978-3-9815370-0-0/DATE13/©2013 EDAA

1.1 Contributions

To address the problem of post-silicon bug diagnosis, we have explored the application of a number of machine learning techniques. First, a hardware mechanism logs compact measurements of observed signal activity over multiple executions of a same test: some passing, some failing. We deploy the data collected from passing testcases to develop a model of expected behavior. This model is then used to evaluate the data from failing testcases, identifying anomalies and, from there, the time and location of a bug. The goal of the system is to:

- Accelerate debugging by automatically identifying the approximate cycle and critical signals involved in a bug.
- Learn correct behavior from passing testcases and differentiate failing behavior using anomaly detection techniques.
- Tolerate non-repeatable executions of the same test, differentiating bugs from noise.
- Develop a solution that can be applied to any on-chip subsystem, without requiring a-priori knowledge of the design.

In particular, we have investigated several machine learning techniques: one based on supervised learning, and another on a one-class learning approach (clustering). We have observed that supervised learning is not a good match for the application of bug-finding, while we have found success with a variation of clustering.

2. MACHINE LEARNING BACKGROUND

Machine learning algorithms build statistical models from examples, which are then used to make predictions when faced with new examples. For instance, in credit card fraud detection, algorithms learn the normal behavior of a customer's banking activity and then monitor for anomalous behavior, which is flagged as fraudulent. The nature of anomaly detection has many similarities to bug detection in hardware designs.

An anomaly detection algorithm relies on input data, called *training data*, to learn the correct behavior of a system. Training data can be *labeled*, often as passing (*positive labels*) or failing (*negative labels*). Each datum is an *example* described by a set of characteristics called *features*. After training, a machine learning algorithm is deployed on unlabeled data, where it is used to *classify* new examples as passing or failing. For example, in credit card fraud detection, training data consists of a customer's past transactions, which are labeled as passing. Each example, a transaction, is described by a number of features that can include the dollar amount, merchant, location, time of day, *etc*. Once the model is trained, each new test execution generates a new piece of data whose label is not known. The goal of the anomaly detection algorithm is to classify the example as either typical or fraudulent.

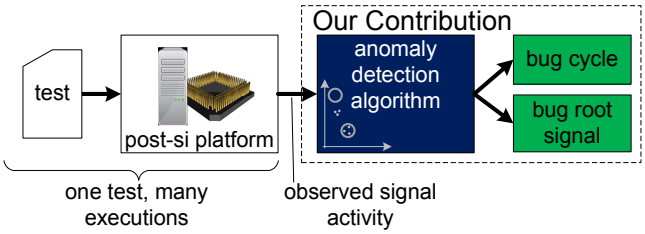

Figure 1: Post-silicon anomaly detection proceeds in two phases. First, multiple executions of a same test are run on the post-silicon platform. Small hardware monitors record a compact measurement of observed signal activity while many test executions pass, and some fail. This data is then analyzed offline by our machine learning algorithm, which builds a model of observed correct behavior from passing testcases, and then uses anomaly detection to identify the time and regions of the design where the failure occurred.

Post-silicon bug diagnosis fits well under the umbrella of anomaly detection. For example, detecting bugs in hardware is analogous to detecting fraudulent behavior in credit card transactions. In both cases, only positive training data is available, and we explain why this is the case in the next section. This challenging learning scenario is called *one-class learning*, and it restricts the applicable algorithms to only learn from positively labeled examples. Furthermore, in fraud detection, examples of fraudulent behavior are rare, just as failing testcases during post-silicon validation are vastly outnumbered by passing cases, especially for difficult-to-reproduce, intermittent bugs.

3. ANOMALY DETECTION ALGORITHM

The goal of our anomaly detection approach is to locate the time of a bug's occurrence, and the critical signals involved during post-silicon validation. Our system operates in two phases. First, it collects data online during test execution, and then it analyzes this data offline to localize the failure (Figure 1). During the collection phase, multiple executions of the same test are run on the post-silicon platform, some of these tests may pass while others fail. A compact measurement of signal activity is recorded for a subset of the design's signals, along with the final test pass/fail result. Measurements are recorded by either existing debug hardware, or by simple custom units. One measurement is collected for each relevant signal and each time step of execution, which comprises a number of cycles. We measure the *fraction of time the signal's value was one during the time step*, an approach that has been shown to be effective [5], and is the *feature* in our application. Once measurements have been collected, we apply machine learning to locate a bug. Each test execution is one *example*, and represents the activity of many signals over many time steps. Our goal is to narrow down this search space to a few signals and execution cycles.

We explored a number of different machine learning approaches. First, we investigated *supervised learning*, where both passing and failing labels are required for training data. While we know the final outcome of each post-silicon test, the origin of the bug is unknown. In the cycles preceding a bug manifestation, the hardware operates correctly, while afterward, its impact causes buggy behavior in at least some signals. As a result, a failing testcase contains both correct behavior (before the bug manifestation), as well as incorrect behavior due to the impact of the bug (after the bug manifestation). Thus, a "failure" on an entire test for the purpose of supervised learning is only a partially correct label. Because the occurrence time of a bug is unknown, we explored other techniques.

3.1 Anomaly Detection Using Clustering

Unlike supervised learning, which requires training data with both positive and negative labels, *one-class learning* requires only a single label. We applied a variation of *clustering,* which groups examples with similar characteristics. When clustering is applied to anomaly detection, the clusters represent correct behavior, while examples that fall outside the clusters could indicate buggy behavior. In our application, the passing label is valid throughout the test, and the label of failing examples is unknown. Thus, our *training data,* was organized by test, and the passing label.

A number of clustering algorithms are present in the machine learning literature [2], among them is *k-means clustering*. The objective of k-means clustering is to assign examples to clusters to minimize the aggregate distance from each example to the center of its cluster. In other words, the algorithm partitions examples into k clusters, minimizing the sum-of-squares distance within clusters.

Figure 2 shows a hypothetical example of clustering applied to two signals during one time step. The X-axis represents the observed activity (feature values) for signal A during the time step, while the Y-axis represents the values for signal B. Green (light gray) dots plot the value pairs measured for these signals. A model for this passing behavior is built using clusters (Figure 2, left), shown by the blue circles on the plot. The number of clusters is a parameter of the algorithm, two in this example.

Next, data collected during failing test executions are plotted in the middle of Figure 2. Some measurements fall inside the clusters, which indicate similar behavior to known-correct test executions. Others fall outside the clusters, and are identified as anomalies. When the number of anomalies exceeds a threshold, a bug is identified (Figure 2, right).

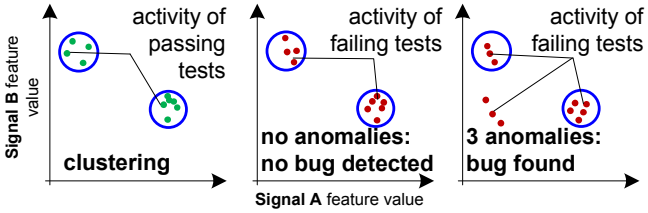

Figure 2: Example of anomaly detection using clustering. Clustering applied to measurements (feature values) from two signals during a single time step, which is a preset number of cycles.

3.2 Scaling to Many Signals and Time Steps

The previous example showed clustering applied to two features, the feature values from two signals during one time step. In practice, the values from a design will include many signals and many time steps. At first, we considered all features at once: all signals during all time steps. In this case, the clustering algorithm considered each signal at every time step to be a dimension, totaling to the number of signals multiplied by the number of steps. There were two problems with this approach. First, since all time steps were considered at the same time, a single analysis comprised activity preceding a bug manifestation, as well as activity following it. Thus, a combination of passing and failing behavior fell under the same failing label, since the testcase failed in the end. The conflation of these passing portions of the test with failing portions of the test led to inaccurate bug localization. The second problem was with high dimensionality, a limiting factor of clustering algorithms.

3.2.1 Two-Step Anomaly Detection

In order to more precisely differentiate the correct behavior that precedes a bug manifestation from the failing behavior that follows

```
    # time detection
 1: for each time step:
 2:   for each module:
 3:     signals = get_signals(module)
 4:     if find_anomaly(signals, time step):
 5:       bug_time = current time step
 6:       goto 7
    # signal detection
 7: for each signal in design:
 8:   if find_anomaly(signal, bug_time):
 9:     bug_signals += signal
10: return bug_time, bug_signals
```

Figure 3: Pseudocode for two-step detection algorithm, which first detects the time of a bug occurrence by analyzing groups of signals with the `find_anomaly` routine (Figure 4). Signals are divided by module in order to limit the dimensionality of the problem. Following a successful time localization, signal detection determines which signals were involved using a second anomaly detection step.

```
 1: find_anomaly(signals, time):
      # calculate clusters
 2:   pass_egs = get_pos_egs(signals, time)
 3:   clusters = cluster(pass_egs)
      # test each failing example
 4:   fail_egs = get_neg_egs(signals, time)
 5:   for each example in fail_egs
        # count anomalies
 6:     if example is outside all clusters:
 7:       anomalies += 1
 8:     if anomalies > threshold:
 9:       return true
10:   return false
```

Figure 4: Pseudocode for clustering, used to detect anomalies. Passing examples are grouped into clusters, modeling correct behavior. Each failing example is then compared to the clusters and those that fall outside are counted as anomalies. When the number of anomalies exceeds a threshold, the function identifies an error.

it, we separated time localization and signal localization. Each time step is analyzed separately, considering all signals within a step together. From this analysis, we can identify which time step presents a sufficient number of anomalies to reveal the occurrence of a bug. Once the time step is identified, a second round of clustering-based anomaly detection identifies the responsible bug signals. The pseudocode in Figure 3 shows an overview of this process. First, time detection proceeds, considering each time step independently (line 1). Next, signals are grouped by module (line 2) and the algorithm performs anomaly detection on the examples from signals in the module (line 4). This process continues through each time step as long as no bug is found. When anomaly detection finds an error, the bug time is recorded as the current time step (line 5) and the algorithm moves to the second phase, ignoring any future time steps (line 6). Signal detection (lines 7-9) examines each signal individually, performing a second anomaly detection, a one-dimensional application of clustering. Upon completion, the algorithm returns both the bug signals and bug time (line 10).

The anomaly detection clustering algorithm (Figure 4) begins with examples from passing testcases (line 2), calculating clusters that model correct behavior. We use an implementation of k-means clustering in a multidimensional space, where each dimension corresponds to a feature (line 3). For example, in the case of the time detection, each signal corresponds to a dimension. Our implementation begins with an initial guess for the location of each cluster's center in this space, called a *centroid*. The guess is chosen at random, and refined during the clustering process. Next, each example is assigned to the nearest centroid, and the sum-of-squares distance of examples mapped to a centroid is computed. The following step calculates a new centroid for each cluster, identifying the point with minimum sum-of-square distance to all the examples in the cluster. At this point, examples are reassigned based on the new centroid locations and then centroids are iteratively recomputed until the algorithm converges. Convergence occurs when the centroid locations no longer change.

After clusters are computed, failing examples are compared to the clusters to determine whether they are consistent with normal behavior. Each failing example is considered (lines 4-5), and the algorithm determines if the example falls within a cluster (line 6). This is accomplished by first computing the radius of each cluster, that is, the distance from the centroid of the cluster to its furthest point. Next, we determine whether each potentially failing example falls within one of the cluster's radii. If an example falls within any cluster, it is not an anomaly. Otherwise, if it is located outside every cluster, then the example is counted as an anomaly (line 7). The algorithm accumulates the total number of anomalies; if the total exceeds a threshold, the function terminates, identifying a bug (lines 8-9). If the number of anomalies remains below the threshold, no bug is flagged (line 10).

3.3 Dimensionality

Experimentally, we found that considering all signals from a large design at once resulted in too high a dimensionality, on the order of 10,000. This is because k-means clustering is NP-hard with respect to the number of dimensions. We first tried using principle component analysis (PCA) [2] to identify the most relevant signals. When we applied PCA to the group of passing signals, the result was a small group of the most noisy signals. This is because noisy signals provide the best means to differentiate among passing test cases (as opposed to quiescent signals). Unfortunately, many potential bug signals were overlooked in this process.

Thus, we developed a heuristic to group signals by module. The number of signals in a module varied widely in the OpenSPARC T2 design, from a few to several thousands, so we capped the number of signals in a group to 500, in order to ensure fast algorithm execution. At first, this approach also had limitations: the number of signals in a group affects the quality of results, thus modules with just a handful of signals were too sensitive to minimal perturbations. Thus, we additionally balanced the partitions with a range of 100 – 500 signals per group. Modules with less than 100 signals were grouped together by parent module, and the signals from modules with more than 500 signals were partitioned into groups of approximately 500. We found experimentally that this solution is effective in mitigating the complexity of k-means clustering, while maintaining good quality of results.

3.4 Anomaly Detection Parameters

Two parameters affect the sensitivity and accuracy of our anomaly detection approach: the number of clusters and the anomaly threshold. During time localization, the number of clusters and the threshold determine when the time step of a bug occurrence is identified. When the number of clusters is small, for example, a single cluster, there is a risk of widely dispersed data causing a very large cluster. Many unnecessary regions are encapsulated by the large cluster: this situation is called *underfitting* (Figure 5a). The consequence

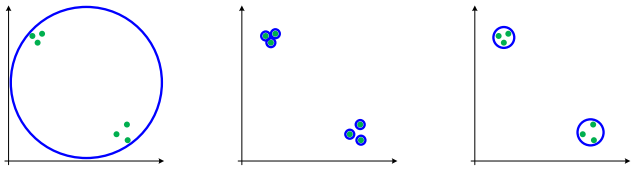

Figure 5: Determining the number of clusters. When the number of clusters is too small (a), underfitting occurs, which can lead to many missed anomalies. On the other hand, too many clusters results in overfitting (b), where a cluster may encompass only a single example. In this case, nearly every new example is considered an anomaly. A good fit balances these two extremes (c).

of underfitting is missed anomalies, which are mistaken for correct behavior. On the other hand, too many clusters may cause *overfitting* (Figure 5b), which in the extreme, leads to a single example for each cluster. The result is an overly sensitive algorithm. Finding a balance, as in Figure 5c, is important to an effective algorithm. We discuss later in the paper how to find this balance experimentally.

In addition to the number of clusters, the second parameter governing the accuracy and effectiveness of our algorithm is the anomaly detection threshold. This is expressed as a percentage of the total failing examples under consideration. A higher threshold leads to a less-sensitive bug-finding algorithm, while low thresholds lead to high sensitivity. In practice, finding the proper threshold can begin with the first passing testcases. An initial threshold can be defined by testing passing executions against other passing executions, gradually increasing the threshold until no bug is found.

3.5 Limitations

While we have found the clustering approach to bug detection to be effective with a set of bugs on the OpenSPARC T2 design, the approach does have a few limitations. First, the algorithm analyzes executions in time steps (a number of cycles), and each time step is considered independently. If one bug causes several subtle perturbations that span multiple time steps, the system may miss it. Bugs that cause only a small delay in one time step suffer a similar problem. Indeed, our approach is most effective for bugs that are detectable within one time step.

A second limitation is training data generation. More training data enables more accurate bug detection, and a possible solution would be to train ahead of post-silicon validation using emulation.

4. EXPERIMENTAL EVALUATION

We evaluated our anomaly-based bug detection algorithm on the industrial-size OpenSPARC T2 design [11], detecting a set of simulated failures. 10 program workloads were used for testing, taken from those provided with the OpenSPARC distribution. Test lengths ranged from about 60,000 cycles to 1.2 million cycles. Table 1 describes the injected bugs, the same bugs used in [5]. The 10 injected errors include functional bugs, where the design logic is modified, electrical failures, where a signal in the design is temporarily altered, and manufacturing faults, which are modeled as stuck-at faults. Faults and failures were injected by forcing a signal in the design for a number of cycles.

First, each test was run 100 times without any bug injection, each time with a different random seed introducing variable memory latency and communication delays. Simulated on-chip hardware collected feature measurements in time steps of 512 cycles (1 time step) for each top-level control signal in the single core (cmp1)

Bug	description
PCX gnt SA	stuck-at in PCX grant
XBar elect	electrical error in crossbar
BR fxn	functional bug in branch logic
MMU fxn	functional bug in mem ctrl
PCX atm SA	stuck-at in PCX atomic grant
PCX fxn	functional bug in PCX
XBar combo	combined electrical errors in Xbar/PCX
MCU combo	combined electrical errors in mem/PCX
MMU combo	combined functional bugs in MMU/PCX
EXU elect	electrical error in execution unit

Table 1: Injected errors in the OpenSPARC T2 design included functional, electrical and manufacturing (stuck-at) failures.

version of the design. There were a total of 41,743 signal bits. The measurements from these 1,000 test executions (10 tests x 100 random seeds) constituted our training data. Next, we injected the bugs, one at a time, running each test with 10 random seeds for a total of 1,000 buggy executions (10 tests x 10 bugs x 10 random seeds). We used this data as unknown examples for identification.

4.1 Bug Localization

The ideal bug localization system identifies a small group of signals shortly after the occurrence of a bug. The evaluation criteria for our machine learning-based approach to bug finding includes the number of signals, as well as the time from bug injection to bug detection. It is also helpful to know whether the exact bug injection signal is among those that were detected. This minimizes the number signals and cycles that a validation engineer must analyze during the debugging process. Furthermore, any statistical system is subject to false positives and false negatives. False positives occur where our solution detects a bug that was not yet injected, and false negatives denote a missed bug. Our goal is to minimize both false negatives and false positives.

In our first study, we examined the types of outcomes for bug detection for each of the 10 bugs. Figure 6 shows the percentage of each of five outcomes, over all 10 testcases. The chart first shows those testcases where the anomaly detection algorithm identified the exact root signal of the bug among those signals detected (dark green at the top of the bars). In other bug/testcases pairs, we

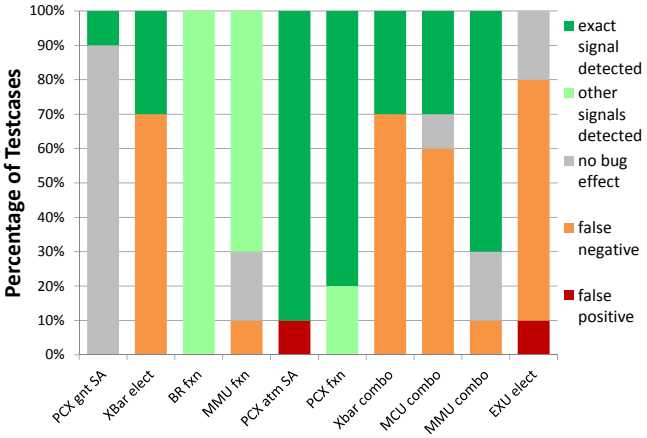

Figure 6: Bug detection outcomes among 10 testcases run on the OpenSPARC T2, showing the percentage of testcases where the exact bug root signal was identified and those where signals exhibiting secondary effects were identified. Additionally, the chart shows bug/testcase combinations where the bug had no effect, as well as false negatives and false positives.

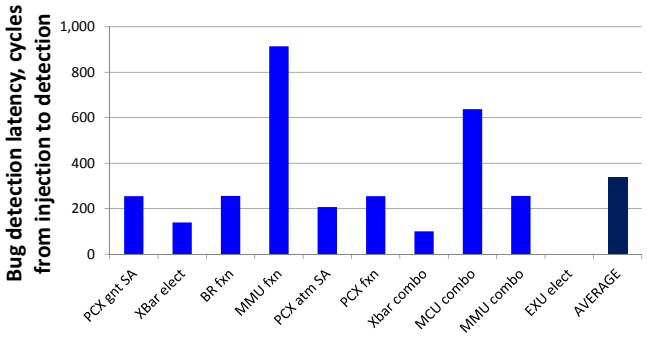

Figure 7: Bug detection latency (cycles), from bug injection to bug detection, averaged over all testcases for each bug.

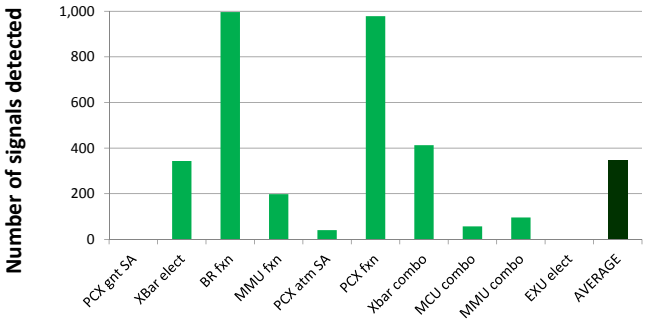

Figure 8: Number of signals detected for each bug, averaged over all testcases. No signals were identified for the `EXU elect` bug, since no bug was detected. A single signal, the exact bug root cause signal, was identified for the `PCX gnt SA` bug.

observed a second outcome, where the bug is detected by its secondary effects on other signals (light green). For two bugs, only secondary effects were observed: `BR fxn` and `MMU fxn`. In both cases, the exact root signal of the bug was not among those available for observation. When a bug was not detected, it was sometimes due to the bug having no effect on a particular testcase, thus the testcase passed (gray). A small number of false positives were observed, where noise was mis-categorized as a bug (red). Finally, our system did not detect the bug in some bug/testcase combinations, shown as false negatives (orange). One bug (`EXU elect`) was particularly difficult to locate, due to effects that did not exhibit significant difference from normal activity. For all other bugs, our system was able to detect them eventually. Furthermore, in all cases where the exact root signal was observable and signals were identified, the exact root signal was among them.

In addition to investigating the outcomes of our approach, we also recorded the bug detection latency. This latency is the number of cycles from bug injection to bug detection. Figure 7 shows the detection latency for each bug, averaged over all testcases exposing the bug. Some bugs were detected quickly, especially those that had high impact on the system. Others were detected after some time, when the accumulation of the bug's impact on the system had become detectable. The `EXU elect` bug was not detected. On average, our system was able to detect bugs very quickly after injection, only 336 cycles, averaged over the bugs. The cycles are measured from the middle of the time step, which was 512 cycles.

The number of signals detected also has an impact on the debugging process. This is determined by the signal detection phase of the machine learning algorithm. Figure 8 shows the number of signals detected, on average, for each bug. No signals were identified for the `EXU elect` bug, since no bug was detected. On the other hand, a single signal was detected for the `PCA gnt SA` bug, which was the exact root signal of the bug. Overall, among the 41,743 signals in the OpenSPARC T2 top-level, the anomaly detection algorithm identified 347, averaged over the bugs. This represents 0.8% of the total signals. Thus, our approach is able to reduce the pool of signals by 99.2%.

4.2 Tuning Parameters

A number of parameters affects the accuracy and effectiveness of our solution. Both the number of clusters and the anomaly detection threshold affect the sensitivity of the approach, including the number of false positives and negatives, as well as the detection latency. The quantity of training data is also important.

The number of clusters was explored in our next study, where we ran bug detection with different numbers of clusters, each time testing all bug/testcase pairs with a threshold of 0.5. For each attempted bug detection, we recorded false negatives and false positives (Figure 9). We observed a trade-off between false negatives and false positives. With a single cluster, the data were under-fit, resulting in many false negatives. On the other hand, when many clusters were used, over-fitting occurred, resulting in false positives. The sum of false negatives and positives was minimized with 2 clusters, which we used in subsequent studies. Different numbers of clusters could be useful with more training data, which would help avoid the case of over-fitting. Finally, we noted that the average detection latency exhibited a decreasing trend as the number of clusters increases.

The anomaly detection threshold is shown in Figure 10, varied from 0.1 to 0.9 with 2 clusters. First, we noted a trade-off in the number of false negatives and false positives. When the threshold was low, the system required very few anomalies to detect a bug, resulting in many false positives. On the other hand, with a high threshold, the system was insensitive, and false negatives dominated. With a threshold of 0.5, the sum of false negatives and false positives is minimized, and we used this value for our other experiments. False negatives and positives were balanced by the detection latency, which was also impacted by the threshold. As the threshold increased, the time from bug detection to bug injection increased.

The quantity of training data also impacted results. Figure 11 plots the number of anomalies over time for one signal, with a bug injected at cycle 50,000. With 10 training examples, the first plot exhibits noise that is nearly as strong as the signal. On the other hand, the second plot was generated using 100 training examples, and exhibits a much better signal-to-noise ratio, shown by the clear delineation of low amplitude noise before the bug and high strength signal after. Thus, we found that with more training data, the system was better able to differentiate noise from buggy behavior.

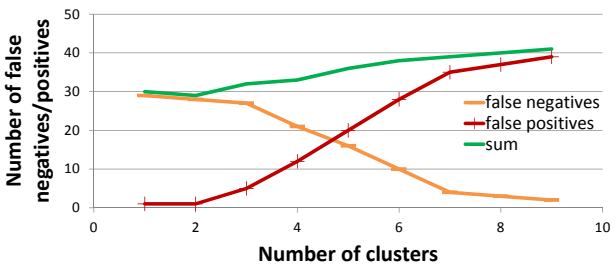

Figure 9: Effect of number of clusters on quality of results, showing the trade-off between false negatives and false positives.

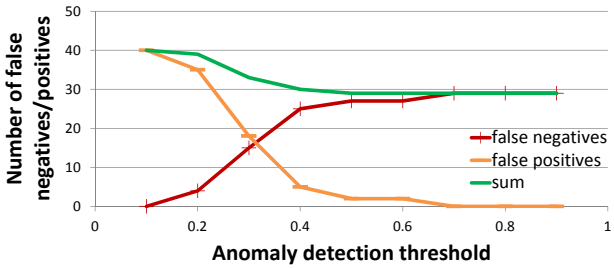

Figure 10: Effect of anomaly detection threshold on results, showing the trade-off between false negatives and false positives.

5. RELATED WORK

The theory of anomaly detection has been studied extensively in the machine learning community [4]. Another application of anomaly detection utilizing similar methodologies to ours is credit card fraud detection [3]. Here, fraudulent activities are rare, and the authors introduce an algorithm that clusters legitimate transactions into groups. If a new transaction falls outside the account's clusters, an anomaly is detected and an alert is raised on the account. This application is similar to the bug diagnosis problem in that the available training data lacks negative examples. Both applications seek to identify behavior that falls outside the norm.

Traditional post-silicon debugging is a predominately manual process, which begins when the first prototypes are available. Engineers run tests on the prototype, checking the outputs for correctness. Often, reproducing the bug is difficult [7], requiring many executions to capture. Next, debugging begins, where on-chip logic analyzers [12] and flexible debugging infrastructures [1] trace signal activity. Engineers use these tools to gather information for manual debugging. Our goal is to automate this process, narrowing the search space of bug times and locations.

Several prior works have addressed the problem of bug diagnosis. At the circuit level, functional bugs have been addressed by recording circuit traces using scan chains, and then comparing failing results against passing results [10]. In the software world, bug reports are automatically analyzed to find errors [8]. During post-silicon debugging, the BLoG/IFRA solution [9] localizes bugs within a processor core. In contrast to these works, we provide a flexible algorithm that is not specific to any particular subsystem.

Guzey, *et al.* [6] applied machine learning to a hardware design for the purpose of increasing the coverage of pre-silicon simulation. They record the bit-vectors of constrained-random test inputs,

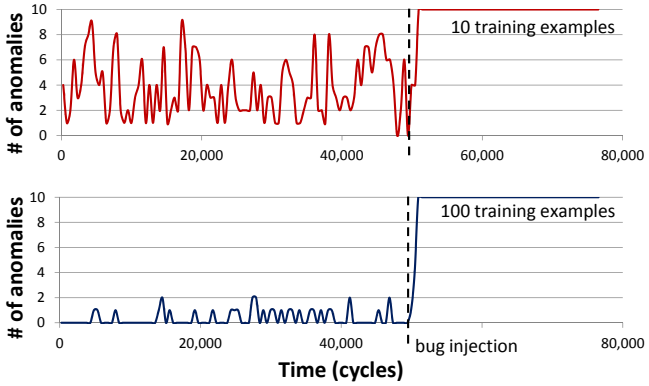

Figure 11: Effect of training data quantity, showing the number of anomalies observed over time for one bug's root signal. The first chart (a) was trained using 10 examples, while the second was trained with 100. There is a much better signal-to-noise ratio in the second, making the time of the bug more clearly identifiable.

building a model of a test's behavior. This model is then used to compute the similarity of a new test to previous tests. While [6] focuses on the similarity of two tests, our approach aims to identify that of corresponding signals in multiple runs of a same test.

BPS [5] proposed an automated approach to bug diagnosis with a specialized algorithm that considers each time and each signal independently. In contrast, this work leverages a formalized algorithm that takes into account the interactions among signals. As a result, we are able to detect the time of a bug occurrence within 336 cycles of its injection, on average, while BPS required 1,273 cycles. Table 2 shows clustering in comparison with BPS [5] and IFRA [9].

	IFRA [9]	**BPS** [5]	**Clustering**
description level	architectural only	✓logic and architectural	
design complexity	single core only	✓whole chip	
type of bugs	electrical	✓functional, electrical, mfg.	
required obervability	pipeline stages	✓high level signals, flexible	
signal interactions?	N/A	no	✓yes
spatial localization	∼10,000 gates (block)	✓75 avg.	347 avg.
temporal localization	✓exact cycle	1,273 cyc avg.	✓336 cyc avg.

Table 2: Clustering compared with IFRA [9] and BPS [5].

6. CONCLUSIONS AND FUTURE WORK

We have presented a machine learning-based approach to post-silicon bug diagnosis. Based on anomaly detection techniques, our algorithm builds a model of correct on-chip signal activity from passing test executions. This model is then applied to detect aberrant behaviors, or anomalies, identifying a bug's time and location.

There are a number of directions for future work in the application of machine learning techniques to post-silicon bug diagnosis. While we found clustering to be effective, we would also like to investigate the use of one-class support vector machines (SVM). Additionally, an exploration of new features used to define examples might improve results.

7. REFERENCES

[1] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller. A reconfigurable design-for-debug infrastructure for SoCs. In *Proc. DAC*, 2006.

[2] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2007.

[3] R. J. Bolton, D. J. Hand, and D. J. H. Unsupervised profiling methods for fraud detection. In *Proc. Credit Scoring and Credit Control*, 2001.

[4] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Computing Surveys*, 41(3), 2009.

[5] A. DeOrio, D. S. Khudia, and V. Bertacco. Post-silicon bug diagnosis with inconsistent executions. In *Proc. ICCAD*, 2011.

[6] O. Guzey, L.-C. Wang, J. R. Levitt, , and H. Foster. Increasing the efficiency of simulation-based functional verification through unsupervised support vector analysis. *IEEE Trans. CAD of ICs and Systems*, 29(1), 2010.

[7] D. Josephson. The manic depression of microprocessor debug. In *Proc. ITC*, 2002.

[8] B. R. Liblit. *Cooperative bug isolation*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 2004. AAI3183833.

[9] S.-B. Park, A. Bracy, H. Wang, and S. Mitra. BLoG: Post-silicon bug localization in processors using bug localization graphs. In *Proc. DAC*, 2010.

[10] P. D. Peter Dahlgren and I. Parulkar. Latch divergency in microprocessor failure analysis. In *Proc. ITC*, 2003.

[11] Sun microsystems OpenSPARC. http://opensparc.net/.

[12] L. Whetsel. An IEEE 1149.1 based logic/signature analyzer in a chip. In *Proc. ITC*, 1991.