ItHELPS: Iterative High-Accuracy Error Localization in Post-Silicon

Valeria Bertacco and Wade Bonkowski
Department of Computer Science and Engineering, University of Michigan
{*valeria, wadeb*}*@umich.edu*

***Abstract*—The increasing complexity of modern digital circuits has exacerbated the challenge of verifying the functionality of these systems. To further compound the issue, shrinking time-to-market constraints place increased pressure on attaining correct devices in short amounts of time. As a result, more and more of the burden of validation has shifted to the post-silicon stage, when the first silicon prototypes of a design become available. This validation phase brings much faster test execution speeds, at the cost of a very limited ability of diagnosing bugs. To further compound the problem, intermittent failures are not uncommon, due to the physical nature of the device under validation.**

In this work we propose ItHELPS, a solution to identify the timing of a bug manifestation and the root signals responsible for it in industry-size complex digital designs. We employ a synergistic approach based on a machine-learning solution (DBSCAN) paired with an adaptive refinement analysis, capable of narrowing the location of a failure down to a handful of signals, possibly buried deep within the design hierarchy. We find experimentally that our approach outperforms the accuracy of prior state-of-the-art solutions by two orders of magnitude.

I. Introduction

The extreme complexity of digital hardware, enabled by the continued evolution of silicon technology, has lead to an unbearable challenge in validating a design's functionality. Indeed, today, design teams are no longer striving to attain the complete functional correctness of a new system, but simply to minimize their exposure to escaped bugs, by validating as many of the design's features and operations as possible, in the limited amount of time available. In this dooming scenario, post-silicon validation – the validation occurring on manufactured silicon prototypes – has become a promising venue to flush out design bugs: testbenches execute several orders of magnitude faster than on pre-silicon software simulators. As a result, in post-silicon, it becomes possible to quickly execute vastly more complex regression tests, reaching deep design states, than it was previously conceivable. Because of these benefits, post-silicon validation has quickly become a key validation methodology in large design houses.

However, validating in post-silicon brings along some challenging aspects: first of all, the scope of validation in this stage is much broader than it was in pre-silicon, since at this stage other physical aspects, such as manufacturing defects and electrical effects, must also be validated along with the design functionality. Second, the observability of the design is minimal, so that diagnosing a bug, once it has manifested, can be extremely difficult, often requiring weeks of engineering effort. Last but not least, because the validation is now carried out on a physical medium (the silicon prototype), additional noise effects are superimposed to the operation of the prototype. This aspect, in turn, may lead to intermittent bugs manifestations when, running a same regression suite multiple times on the same prototype, leads to observing the bug (that is, the incorrect output) only occasionally. This latter problem, a bug that cannot be consistently reproduced, is often unsurmountable, since regression suites in post-silicon are extremely complex, reaching states that are buried deep in the design, while physical effects and other software processes make it almost impossible to produce multiple identical executions. Hence, to address intermittent bugs, it is necessary to devise diagnostic solutions that do not rely on the ability to reproduce the bug.

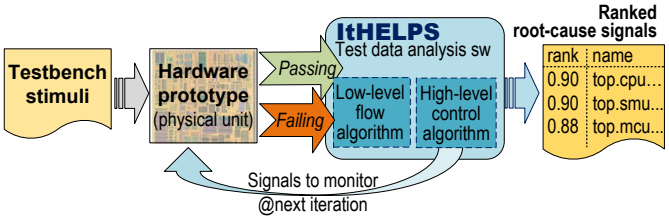

Fig. 1. **ItHELPS overview.** ItHELPS applies an iterative refinement flow to the design under validation: first it elaborates on the data gathered by focusing on a handful of signals over multiple runs of a same test – some passing and some failing. Then, it generates new sets of signals to monitor for subsequent batches of test runs. In the end, ItHELPS provides a ranked list of signals, with the top ranked ones being the most likely root-cause of the bug.

Contributions. To alleviate the challenges outlined above, this work proposes **ItHELPS**, an **It**erative **H**igh-accuracy **E**rror **L**ocalization solution in **P**ost-**S**ilicon. ItHELPS employs an iterative machine-learning based approach capable of identifying the root cause of a test failure, whether functional or of other nature, within a handful of signals, 3 on average in our experiments. ItHELPS is an automated solution, capable of localizing failures by mining the activity of the prototype when running the same regression test multiple times. The signal activity information that ItHELPS requires for this purpose is very simple and many post-silicon frameworks are already equipped to gather it [1]. Note that for ItHELPS to attain results, it is not necessary that the outcome of the test be consistent (passing or failing): indeed, if that were the case, other mainstream validation solutions could be deployed [2,3,4]. Compared to other solutions in this space, ItHELPS provides significant advantages in (i) the quality and accuracy of its results and (ii) its unique ability to diagnose bugs whose root cause is buried deep in the design hierarchy. To attain these results, ItHELPS leverages at its core a powerful machine-learning algorithm, DBSCAN [5], coupled with a novel, iterative refinement search flow.

II. Overview

Figure 1 presents an overview of ItHELPS's flow. The design under validation executes the same test multiple times: some executions are successful (*passing*), that is, the checkers at the end of the test do not detect any bug, others fail (*failing*). ItHELPS consists of a software analysis framework capable of analyzing the data in these two sets and pinpointing the time and signals where the executions first started to diverge, thus diagnosing the bug down to a small time window and a handful of candidate root-cause signals.

We assume that the design is equipped with on-chip sensors [1], as it is common in many post-silicon setups, capable of monitoring a handful of signals at a time (16-32 is typical) over the entire execution of the test. To minimize the performance cost of transferring signal values off-chip, this data is often compressed on-the-fly, transforming the sequence of signal values, into a sequence of "signatures". ItHELPS provides an iterative refinement approach, whereby it analyzes the signatures gathered during a set of executions and, based on its findings, it selects a new set of signals to monitor for the next batch of test runs. In each of its analyses, our solution strives to determine if the behavior of any of signals that were monitored was highly divergent between passing and failing runs. If that is the case, the signal is probably close to the region where the bug first manifested; thus, it was strongly perturbed. ItHELPS then selects a new pool of signals in the vicinity of the diverging one, in the hope to find other signals diverging even more markedly, and triggers a new batch of runs. This iterative approach is what enables ItHELPS to attain such high quality of results compared to prior solutions [6,7].

A. Signal Tracing

Signal tracing has become a mainstream component of post-silicon methodologies as a way of gaining observability over a target portion of a design under validation. Techniques such as scan-chains and JTAG interfaces [8,9], have been deployed since the early days of post-silicon validation. A more recent approach that is gaining adoption is based on multiplexing signal tracing [1], which entails the use of built-in multiplexor tree-structures to select a small number of internal signals from a rich pool, and route them to the "surface" of the design for off-chip transfer. In our solution, we assume this type of signal monitoring approach, where only a handful of signals can be inspected during each batch of test executions; and we designed ItHELPS to be robust to this limitation.

The sequence of signal values traced are then compressed into signatures [6,10]. Similarly to [6], we assume here that the signature is calculated simply by counting the fraction of cycles during which the signal assumes the logic value 1 during a fixed length of the execution. Note, however, that ItHELPS is agnostic to the specific signature mechanism, if any: a valuable aspect is that the compression provides a high-pass filter on the data, so that small signal differences are amplified into large signature variations. Note that the counting window length for the signature computation can be varied based on the time granularity desired for the diagnosis.

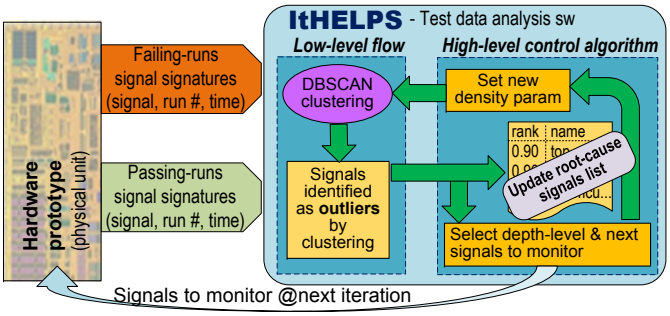

Fig. 2. **ItHELPS components.** Our solution includes a machine-learning algorithm, DBSCAN, and the iterative refinement module, responsible for selecting the next set of signals to be analyzed and the new density parameter for DBSCAN. This module also maintains the list of candidate root-cause signals, ranked by the highest density value at which they could be flagged.

III. Analysis Software

ItHELPs consists of two main components: the machine-learning algorithm, DBSCAN, responsible for identifying "outlier" signals, and a novel, adaptive refinement module, which determines which set of signals should be monitored, and the density parameter value that DBSCAN should use. This latter module is also responsible for maintaining the list of candidate root-cause signals, ranked by the associated density parameter, and for terminating the analysis, when improvements have dwindled. ItHELPS considers two sets of signatures, one from executions that passed and one from those that failed. Moreover, each signature in a set is indexed by which time window of execution and which signal it corresponds to. For each time window and signal we collect multiple signatures, one for each execution of the test.

A. Identifying outliers with DBSCAN

DBSCAN, Density Based Spatial Clustering for Applications with Noise [5], is an unsupervised density-based clustering algorithm. It first considers all the passing signatures from a batch of test runs and creates clusters based on their distribution, analyzing one time window of execution and one signal[1] at a time. It then includes the signatures from failing runs and counts the number that fall outside the clusters built in the previous step. If this number raises above a user-defined threshold, then the corresponding time window and signal is flagged as potentially being at the root of the bug. The signal identified in this process is associated to the density parameter used in the analysis and added to the "root-cause signals list", which is sorted by decreasing density value. DBSCAN requires one *density* parameter to generate the clusters. The parameter sets a limit on *how far a point can be from another to be in its same cluster*, and it is controlled by our refinement module.

For the purposes of our goals, DBSCAN is more powerful than other popular clustering algorithms (*e.g.*, k-means) because it is flexible in the number of clusters it builds and it is capable of ignoring noise that doesn't fit into any of the clusters. Both these aspects are particularly valuable for us because (i) each

[1]DBSCAN could analyze many signals at a time, but we found that single signal analyses provide more accurate results.

design/test pair requires unique clustering criteria for its signals' activity: some tests cause signals to have naturally occurring high variations, while others have very uniform activity patterns. Moreover, (ii) a certain amount of variation between different executions is always present, and it is important that the clustering algorithm is capable of identifying and disregarding it. Note that DBSCAN cannot operate with varying density clusters, thus, our solution provides a specific density parameter for each clustering task, so to provide adaptivity over the analysis. Finally, note also that because of the small number of signals monitored during each batch of test runs (corresponding to one analysis step), the execution of DBSCAN is quite fast and, contrary to mainstream machine-learning applications, it does not require to overcome algorithmic complexity challenges. In selecting DBSCAN, we evaluated a number of machine-learning solutions and we found that it was the most flexible in adapting to variations in bug manifestation patterns. We further strengthened the robustness of our diagnosis solution by devising the iterative refinement approach discussed below.

B. Iterative refinement

The iterative refinement component of ItHELPS forces our software to co-execute with the post-silicon validation platform: after a batch of test runs is completed, ItHELPS clusters the data and identifies potentially buggy signals; then, based on those findings, it determines the new set of signals to monitor in the next batch and the new density parameter. Finally, it returns the control to the validation platform.

Our refinement algorithm traverses the design hierarchy with a depth search approach. It first investigates signals in the top-level module of the design. When a signal is flagged, it analyzes its surrounding region in great detail, and then goes deeper into modules connected with the signals identified. Whenever a portion of the search completes, it backs up to the prior level and continues the search.

The algorithm uses three different signal-selection modes, based on the phase in which it is operating. During the *uniform search phase*, when there is no flagged signal yet, signals to be monitored are selected uniformly across the candidate modules. For instance, at the beginning of the analysis, the candidate module is the top-level module of the design. When a signal is flagged by DBSCAN, the algorithm enters the *targeted search phase*, and switches to focus on the surroundings of the suspected signal, by selecting other signals from the same module. Once all signals in that module have been considered, our algorithm enters the *refinement phase*, where it reconsiders all the flagged signals from the same target module, but with an increased density value, so that it can be more selective in its ranking. It iterates this process and increases the density value until no signal can be flagged by DBSCAN anymore. This phase allows ItHELPS to update the root-cause signals list with accurate rankings, representing the highest density value at which each signal was found buggy. When the analysis completes in one module, ItHELPS strives to determine if the root-cause of the bug lies in a lower-level module. It does do by repeating the process, starting with the uniform search phase, but considering signals from lower-level module(s) that are connected to previously-identified buggy signals.

Our algorithm has two terminating conditions: (i) reaching sufficient coverage of the top-level design module (set through a user-defined threshold), or (ii) finding a signal that provides very high confidence of being the root cause of the bug.

IV. Experimental Evaluation

We evaluated ItHELPS on an Oracle's OpenSPARC T2 64-bit multi-threaded processor [11], targeting the analysis to one of the cores. We emulated the post-silicon executions on a software simulator, injecting a variety of functional (*fxn*), electrical (*elect*) and manufacturing bugs (*SA*), either in the design or directly in the simulation. The bugs' names indicate the module where they manifest and the type of bug. For our tests, we used testcases distributed with the design. ItHELPS was implemented in Python and ran on 8-core Intel Xeon CPUs with 24GB of memory. Furthermore, we eliminated data signals from the search space and excluded noisy signals through a common-mode rejection filter applied on a per-test basis.

We initialized the density parameter to 0.10 (the range is $[0, 1)$), and we set to 2 the minimum number of points that must be within "density" distance in order to form a cluster. We chose this value because we used 10 passing runs to do the clustering, so we only had 10 points in each DBSCAN clustering task, and we wanted to maintain the option of forming multiple clusters. Our time windows were set to 512 clock cycles and, in all cases, we were able to localize the bug within a single time window. Note that we chose this value to strike a trade-off between detection time accuracy and time to collect all the signatures in our evaluation. We believe that ItHELPS would perform equally well with finer granularity time windows.

	blimp_rand	fp_addsub	fp_muldiv	isa2_basic	isa3_asr	isa3_wndw	ldst_sync	mpgen	n2_lsu	tlu_rand	COMBINED
PCXgnt SA	1 / 0.98	nobug	nobug	nobug	nobug	nobug	nobug	nobug	nobug	nobug	
Xbar elect	1 / 0.98	2 / 0.98	1 / 0.38	0	1 / 0.98	0	0	1 / 0.70	0	2 / 0.98	
PCXatm SA	1 / 0.98	2 / 0.98	2 / 0.98	1 / 0.98	1 / 0.98	1 / 0.98	0	0	0	1 / 0.98	
PCX fxn	1 / 0.94	3 / 0.98	2 / 0.98	2 / 0.98	1 / 0.12	1 / 0.24	1 / 0.98	12 / 0.98	nobug	1 / 0.98	
MMU fxn	0	3 / 0.90	2 / 0.90	nobug	0	nobug	2 / 0.90	2 / 0.90	0	2 / 0.90	
DEC elec	1 / 0.70	0	0	0	1 / 0.22	1 / 0.22	4 / 0.70	0	0	2 / 0.50	

Fig. 3. **ItHELPS root-cause signal detection.** The table reports how many signals ItHELPS flagged as buggy and the corresponding top density value, for each test / bug pair. The green shading indicates that the pool of signals included the correct root-cause signal. The far right column indicates whether the bug was correctly diagnosed by ItHELPS over all tests.

The table of Figure 3 reports the findings of the analysis for a number of test / bug pairs. In each slot of the table we report the number of signals in the top rank of the root-cause signals list, and the corresponding density value. The slots marked in green correspond to analyses where ItHELPS was able to identify the correct root-cause signal. The slots with a 0, indicate that ItHELPS could not flag any signal, while `nobug` means that the bug did not manifest and the test completely always correctly. Note how in every case ItHELPS flags only a handful of signals, in contrast with prior solutions [6,7], which produce hundreds of candidate signals, each to be checked to

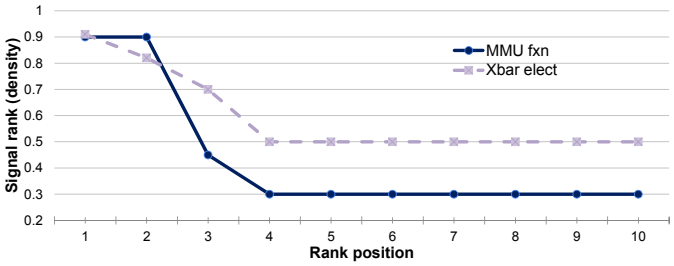

Fig. 4. **Root-cause signals list.** The figure plots the rank (density parameter) for the top 10 signals included in the root-cause signals list when analyzing two bugs, *MMU fxn* and *Xbar elect* – averaged over all testbenches. Note how quickly the density drops after just three signals.

confirm the diagnosis. Note that the first four bugs are injected at the top level, while the next two are injected deeper in the design hierarchy. The computation time required by ItHELPS increases with the depth of injection of the bug, because of the additional coverage required for each module and level visited by the algorithm. Indeed, we considered two additional bugs not reported in the table, a functional bug related to the branch unit (*BR fxn*), and a manufacturing bug on the instruction level-2 cache (*I2$ SA*): in both cases the root-cause signal lies in the third level of the design hierarchy, and ItHELPS was able to precisely pinpoint it. For instance, in the latter bug, it identified a set of 10 signals with a rank of 0.90 or above, which included the bug.

To gain further insights on how selective ItHELPS is in its diagnosis, Figure 4 charts the rank corresponding to the top 10 signals in the root-cause signals list for two sample bugs, *MMU fxn* and *Xbar elect* (averaged over all testbenches): note how the rank drops quickly from 0.9 to 0.4-0.5 within just four list entries. In most cases, the root-cause signal of the bug is precisely the top-ranked signal(s) reported by ItHELPS.

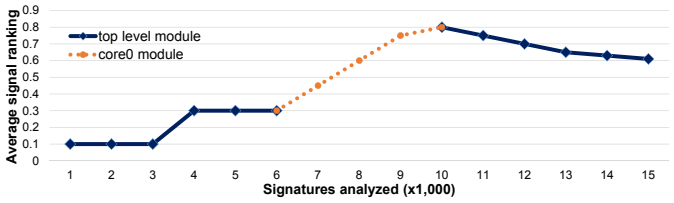

Fig. 5. **ItHELPS diagnosis progress.** The plot charts how the average rank of the signals in the root-cause list grows throughout the analysis.

Finally, we present an example on how the signal ranking evolves over time. For this example we considered again the *MMU fxn* bug. This bug affects a signal that is defined one level below the top module, within the memory management unit of the *core0* processor. Figure 5 plots the average ranking of the signals included in the root-cause signals list over the progression of ItHELPS analyses. Note how, at first, the density parameter of the signals included is at 0.10, the initialization value.When ItHELPS stumbles upon a promising outlier signal (3,000 analyses), it focuses in its surroundings, quickly finding a number of highly ranked signals. It then moves into the second-level module that is connected to the signals flagged so far, and works in this region, quickly raising the average rank by adding more signals closely related to the bug. At 10,000 analyses, this level's search is complete, and ItHELPS has found the root-cause signal. It then resumes the inspection at the top level to attain good coverage over its signals.

V. Related Work

Post-silicon bug diagnosis has been studied in several recent works [2,3,4,6,7], leveraging formal analysis techniques and algorithms for signal tracing selection. A few of them address the diagnosis problem in the presence of intermittent bug manifestations, leveraging both statistical analysis and machine-learning algorithms. In particular, [7] attacks the problem with a popular clustering algorithm, *K-means*. However, the lack of adaptivity of K-means has led the authors to a much weaker localization ability compared to ItHELPS, with hundreds, up to a thousand candidate root-signals. The comparison of passing against failing test results has also been leveraged to detect functional bugs in hardware designs, *e.g.*, [12] records circuit traces using scan chains, and then applies the comparison to detect discrepancies. Finally, other authors have deployed machine-learning solutions in the context of hardware design problems. An example is [13], where the authors record the stimuli in constrained-random tests and use them to build a model of a test's behavior. This model is then used to discern how novel the test is w.r.t. tests already included in a regression, with the ultimate goal of boosting coverage.

VI. Conclusions

We presented ItHELPS, an effective solution to diagnose intermittent post-silicon bugs with high accuracy. ItHELPS leverages a powerful machine-learning technique (DBSCAN) paired with an iterative refinement algorithm to diagnose post-silicon bugs to a handful of candidate root-cause signals, possibly buried deep in the design hierarchy.

Acknowledgements. This work was partially supported by NSF grant #1217764 and C-FAR, within STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

[1] X. Liu and Q. Xu, "On multiplexed signal tracing for post-silicon validation," *IEEE Trans. CAD*, vol. 32, no. 5, 2013.
[2] F. DePaula, M. Gort, A. Hu, S. Wilton, and J. Yang, "Backspace: formal analysis for post-silicon debug," in *Proc. FMCAD*, 2008.
[3] K. Basu, P. Mishra, and P. Patra, "Efficient combination of trace and scan signals for post silicon validation and debug," in *Proc. ITC*, 2011.
[4] C. Zhu, G. Weissenbacher, and S. Malik, "Post-silicon fault localisation using maximum satisfiability and backbones," in *Proc. FMCAD*, 2011.
[5] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proc. KDD*, 1996.
[6] A. DeOrio, D. Khudia, and V. Bertacco, "Post-silicon bug diagnosis with inconsistent executions," in *Proc. ICCAD*, 2011.
[7] A. DeOrio, Q. Li, M. Burgess, and V. Bertacco, "Machine learning-based anomaly detection for post-silicon bug diagnosis," in *Proc. DATE*, 2013.
[8] D. Josephson and B. Gottlieb, *Advances in Electronic Testing: Challenges and Methodologies*. Springer, 2006, ch. 3.
[9] G. Rootselaar and B. Vermeulen, "Silicon debug: Scan chains alone are not enough," in *Proc. ITC*, 1999.
[10] E. Anis and N. Nicolici, "Low cost debug architecture using lossy compression for silicon debug," in *Proc. DATE*, 2007.
[11] "Oracle OpenSPARC," http://opensparc.net/.
[12] P. Dahlgren, P. Dickinson, and I. Parulkar, "Latch divergency in microprocessor failure analysis," in *Proc. ITC*, 2003.
[13] O. Guzey, L.-C. Wang, J. Levitt, and H. Foster, "Increasing the efficiency of simulation-based functional verification through unsupervised support vector analysis," *IEEE Trans. CAD*, vol. 29, no. 1, 2010.