Effective Post-Silicon Failure Localization Using Dynamic Program Slicing

Ophir Friedler, Wisam Kadry, Arkadiy Morgenshtein, Amir Nahir, and Vitali Sokhin
IBM Research - Haifa, Israel
Email: {ophirf,wisamk,arkadiym,nahir,vitali}@il.ibm.com

***Abstract*—In post-silicon functional validation, one of the most complex and time-consuming processes is the localization of an instruction that exposes a bug detected at system level. The task is particularly difficult due to the silicon's limited observability and the long time between a failure's occurrence and its detection.**

We propose a novel method that automates the architectural localization of post-silicon test-case failures. Our proposed tool analyzes a failing test-case, while leveraging the information derived from executing the test on an Instruction Set software Simulator (ISS), to identify a set of instructions that could lead to the faulty final state. The proposed failure localization process comprises the creation of a resource dependency graph based on the execution of the test-case on the ISS, determining a program slice of instructions that influence the faulty resources, and the reduction of the set of suspicious instructions by leveraging the knowledge of the correct resources.

We evaluate our proposed solution through extensive experiments. Experimental results show that, in over 97% of all cases, our method was able to narrow down the list of suspicious instructions to under 2 instructions, on average, out of over 200. In over 59% of all cases, our method correctly reduced a test-case to a single faulty instruction.

I. Introduction

The primary goal of the post-silicon validation effort is to detect, analyze, and find the root cause of design functional and electrical bugs that escaped the pre-silicon verification effort. Despite continuous improvements in pre-silicon verification technologies, both formal and simulation-based, the role of functional validation at the post-silicon stage continues to grow.

Fails debugging is considered to be a major challenge in the post-silicon validation process, and has, accordingly, received significant attention from the research community over the years [12], [14], [9]. The combination of two primary factors make debug a complex problem: the limited observability into the state of the design and the duration of time between the origin of the fail to the first time it is observed.

The focus of this paper is the automation of the *architectural localization* of post-silicon failures. Specifically, we focus on the localization of functional data flow bugs. Such errors occur when data is corrupted during one of the instruction execution phases, for example, in a case in which an instruction fails to write its results to one of its outputs.

Architectural localization is the first step in debugging a post-silicon fail[1], and is aimed at locating the instructions in the test-case where incorrect DUT behavior propagated to the architectural level. Based on the data gathered in the architectural localization phase, i.e., the identity of the suspicious instructions, the location of these instructions in memory, and the operands they accesses, etc., the DUT's hardware debug logic can be configured to trace specific signals [13], facilitating the root cause of the bug.

We propose to use the Instruction Set Simulation [8] (ISS, sometimes called *golden model* or *reference model*) as a vehicle to explore fail reasons and obtain observability into the architectural changes triggered by the failing test-case.

Our method calls for re-running the failing test-case on the ISS[2]. By running the test-case on the ISS, we can determine, for every architectural resource, whether the value it had at the end of the execution of the test-case on the DUT was correct or faulty. We therefore partition all architectural resources into two sets: *correct resources* and *faulty resources*. In addition, based on the intermediary architectural values, as observed on the ISS, we construct a *dependency graph* describing the changes to these resources throughout the execution of the test-case and their dependency on one another.

Leveraging dynamic slicing techniques [7] and based on the set of faulty resources, we traverse the dependency graph to find a subset of the test-case instructions that affect these resources. We term this subset of the test-case the *program slice*. Since the ISS models the dependencies among resources as they are created by the instruction, the program slice is guaranteed to hold the instruction that the DUT failed to execute.

The program slice includes all the instructions in the test-case that affected the faulty resources. However, these instructions may also affect additional resources. We propose a heuristic that, based on our knowledge of the correct resources, removes some suspicions instructions from the program slice. This heuristic relies on a simple rule: if an instruction affects some resource that holds a correct value at the end of the test-case, then the instruction must have been executed successfully. Therefore, our heuristic would remove such an instruction from the suspicious instructions list.

[1]We coined the term *architectural localization* referring to instructions in the test-case, as the basic term *localization* often refers to the process of finding the faulty unit in the design [19], [18].

[2]Some industrial ISSs run sufficiently fast to enable re-running a very long test [1], while other tools rely on sophisticated seed generation techniques to provide that capability [4].

978-3-9815370-2-4/DATE14/©2014 EDAA

To minimize errors when analyzing instructions that write to multiple resources, we propose two flavors of our heuristic. Our first version, the *basic* version of the heuristic, would apply the above rule if at least one resource exists that was affected by the instruction and that is correct at the end of the test-case. The second version, the *conditional* version, of the heuristic would determine that an instruction is no longer suspicious only if *all* affected resources are correct at the end of the test-case.

We validated our approach through error injection experiments. Our results indicate that in over 59% of all cases, our method was able to single out the problem, that is, to correctly reduce a test-case of over 200 instructions to a single faulty instruction. In over 97% of all cases, our method was able to narrow down the list of suspicious instructions to under 2 instructions, on average, out of over 200.

Contributions. We propose an automated architectural localization analysis of post-silicon test-cases by building a dependency graph based on running the failing test-case on an ISS. We describe an algorithm for identification of suspicious instructions, by performing dynamic slicing of the dependency graph based on the list of faulty resources at the end of the test-case. We present a heuristic for reducing the set of suspicious instructions leveraging the list of correct resources, while retaining high accuracy of error localization. We present extensive simulation results that demonstrate the effectiveness of the proposed localization technique, based on error injection in hundreds of randomly generated test-cases. We used a PowerPC™ ISS to implement and validate the proposed algorithms.

The rest of the paper is organized as follows: In Section II we review related work. We describe our method of localizing architectural level failures in Section III. We present our experimental results in Section IV. In Section V, we conclude the paper and present future work.

II. Related Work

Post-Silicon validation is an extensively researched topic, and much like this paper, the majority of work focuses on different aspects of debugging fails. Most of the prior work focused on constructing different mechanisms in the form of added logic to the design to collect data to enable a more efficient debug process. Our proposed solution is a "pure software" one, such that we assume nothing about the structure and internals of the design under test.

In [12], the authors propose a technique with which the validation engineer can reconstruct a long trace of the design through repeated execution. In [18], the authors suggest a mechanism tailored for processors that, once dumped from the design, enables a good understanding of what occurred in the pipeline when the processor triggered a fail. Multiple other papers, such as [16], [22] and [6] are of a similar nature and propose signal-selection techniques, debug logic structures, and analysis methods to assist the post-silicon debug process.

Some works, such as [14], [2] and [10], take this approach a step further and augment the design with logic that has checking capabilities.

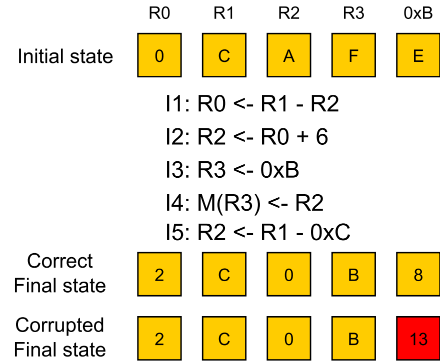

Fig. 1: Test-case example

In [15] and [17], the authors propose a variety of techniques of mutating test-cases, without any change to the logic of the design under test, to localize the fail and ease the debug efforts. We have a similar goal, namely architectural localization. However, our method requires neither changing the test-case nor running it (or mutated versions of it) on the buggy hardware.

Finally, dynamic slicing techniques were initially suggested 30 years ago [23], [5] and are continuously being refined and improved to date [7]. However, these methods have been in use for the purpose of localizing errors in software development, assuming the underlying hardware works well. Similar techniques have been proposed to assist in the hardware development process [20], [21]. These, however, refer to slicing of the RTL code while we propose to conduct dynamic slicing of the test-case. This is the first time these methods are put to use to assist the post-silicon debug process.

III. Localization Algorithm

This paper focuses on the localization of data flow errors. Data flow errors occur when data is corrupted during one of the instruction execution phases. Some examples of the data flow error are: an instruction reading a wrong value from one of its inputs, an instruction failing to compute the right output, or an instruction writing a wrong value into one of its outputs. The faulty data may propagate and contaminate other resources along the test-case execution flow, making the manual error localization process difficult and time consuming.

Let us consider the test-case example shown in Figure 1. The test-case includes five instructions ($I1...I5$) that use four registers ($R0...R3$) and one memory location at address $0xB$. Running the test-case with the initial state of $R0 = 0x0, R1 = 0xC, R2 = 0xA, R3 = 0xF, 0xB = 0xE$ on a bug-free processor should always produce the correct final state of $R0 = 0x2, R1 = 0xC, R2 = 0x0, R3 = 0xB, 0xB = 0x8$. However, let us assume that $I2$ erroneously writes the value 13 instead of the value 8 to $R2$ during the test-case execution. In this case, the wrong value will propagate until it will be written in memory at address $0xB$ and will appear at the final state of a test-case run.

The correct final state can be obtained by running the same test-case on an ISS. In our example, the memory resource $0xB$ will be identified as faulty based on the comparison of the final states of original test-case run on hardware and the ISS run. In this work, we propose a tool that analyzes a failing test-case while leveraging the information derived from comparison of DUT and ISS runs, to identify a set of instructions that could lead to the faulty final state.

The proposed failure localization process consists of three major steps. First, we build a resource dependency graph based on execution of the test-case on the Instruction Set Simulator. Next, we determine a program slice of instructions that influence the faulty resources. Finally, we leverage the knowledge of the correct resources to reduce the set of suspicious instructions. This is done by traversing the dependency graph and marking the instructions that are related to the correct resources.

Figure 2 depicts a resource dependency graph built by running the test-case in Figure 1 on the ISS. Nodes in the graph are grouped in layers, with each layer representing a consistent architectural machine state. The first layer represents the initial state, $S0$, and the last layer represent the final state. Execution of each instruction moves the processor to the next architectural state. For example, execution of $I2$ moves the machine from $S1$ to $S2$. For the sake of clarity, at intermediate states we depict only the resources that were modified by the related instruction.

Resource B is *dependent* on resource A if an instruction I exists with input A and output B. We represent dependencies as edges between resources at two different layers of the dependency graph. For example, since $I2$ reads from $R0$ and writes to $R2$, $R2$ depends on $R0$.

The dependency graph is built based on the test-case execution on the ISS. The ISS eliminates any loops initially present in a test-case by unrolling the loops into a series of instructions per iteration. Hence, by construction, the dependency graph does not contain any loops.

In the second stage, we create a program slice of instructions that affect the faulty resources. The faulty architectural resources are determined by running the test-case on the ISS and comparing values of architectural resources with the execution of the test-case on the DUT. To build a program slice, we traverse the dependency graph using Depth First Search (DFS) [11], starting from each $faulty$ resource node at the final state (bottom layer). During the traversal we mark all the visited nodes at the intermediate states, as *suspicious*. A *program slice* is a set of all instructions that have at least one output resource marked as $suspicious$. Figure 3 depicts such graph traversal. The traversal starts at the final state of memory located at address $0xB$. All marked nodes are shown in red (dark). The corresponding program slice consists of $I1, I2, I3, andI4$. As shown, the set of suspicious instructions contains the faulty instruction $I2$. Our results, described in Section IV, show that building a dynamic program slice substantially reduces the search space as compared to the complete test-case.

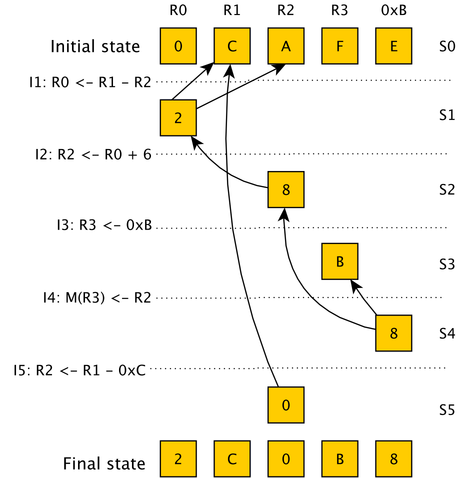

Fig. 2: Resource dependency graph: each layer represents a machine state with edges showing the resource dependency according to inputs and outputs of related instructions

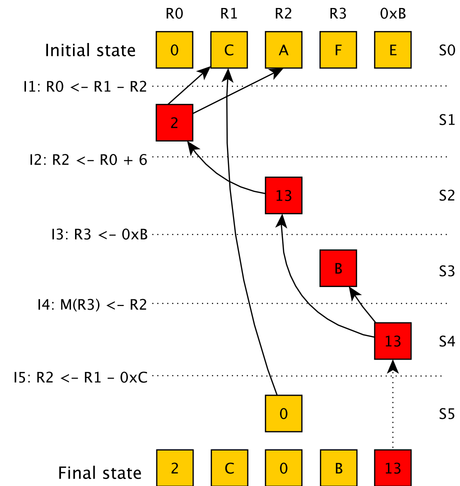

Fig. 3: Dynamic slice: suspicious nodes are marked during the DFS traversal from faulty resources in the final state

The dynamic program slice includes all the instructions in the test-case that affected the faulty resources. However, these instructions may also affect additional resources. In the third stage, we leverage the knowledge of the *correct* resources from the ISS run by employing a *justification* heuristic that further reduces the set of suspicious instructions.

Consider $I3$ from the example above. At the end of our slicing stage, $I3$ is part of the dynamic slice. However, had the DUT erred in executing $I3$, we expect that $R3$, the output of $I3$, would have remained corrupt until the end of the test-case. Since we know $R3$ holds a correct value at the end of the test-case, we can remove $I3$ from our suspicious instructions list.

We implement this heuristic by running DFS from each *correct* resource at the final state. We mark each node reached by the DFS traversal. If all resources associated with an instruction were marked, we remove this instruction from the suspicious instructions list.

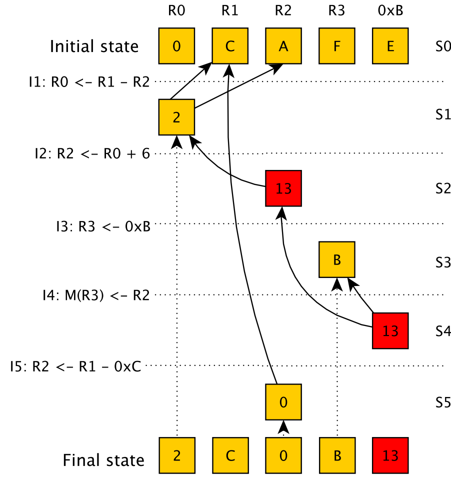

Fig. 4: Justification heuristic: some of the instructions are cleared during the traversal from correct resources in the final state

Figure 4 presents the dependency graph with nodes marked by the justification traversal shown in the original yellow color. Now the reduced set of suspicious instructions contains only I_2 and I_4. The remainder of the previously suspected instructions are cleared by the justification traversal from the correct resources $R0$ and $R3$. Note that the faulty instruction I_2 is still in the final suspicious list.

All modern architectures include instructions that write to multiple outputs. When such an instruction is badly executed by the DUT, the nature of error propagation heavily depends on the nature of the bug. For example, the DUT may err in reading a value from memory, and the error may propagate to *all* of the instruction's output. On the other hand, a case may occur in which the DUT errs in writing to *one or more* of the instruction's outputs, but not to all. We assume that the data we have does not contain any knowledge of the nature of the bug in the DUT. Thus, we propose two flavors of justification heuristic, in which instructions are cleared based on analysis of correct output resources.

The *basic* justification heuristic, as shown in Figure 4, assumes that errors propagate to *all* instruction outputs. Therefore, if at least one of the instruction outputs is not suspicious, the instruction is removed from the suspicious instructions list. This, however, may lead to excessive reduction of the program slice and to the erroneous clearance of faulty instruction in which only part of the outputs are corrupted.

The *conditional* version of the heuristic assumes that an error may propagate to *some* but not all of the instruction's outputs. Therefore, only if all of the instruction's outputs are not suspicious, the instruction is removed from the suspicious instructions list.

The *conditional* justification heuristic allows us to efficiently reduce the size of the program slice, without compromising the failure localization accuracy. Note that in cases of single-output instructions, the definitions of the two versions of the heuristic converge, and they provide the same output.

In this section, we formally describe our approach. Let $I = (I_0, I_1, I_2, ..., I_m)$ be an ordered set of instructions executed on machine M. Let $R = \{R_0, ..., R_n\}$ be a set of resources of machine M, in which each resource R_i is either a register or a memory location.

Algorithm 1 Generation of dependency graph

1: Input: $I = (I_1, ..., I_m)$, $R = \{R_0, ..., R_n\}$
2: $i \leftarrow 1$
3: $V \leftarrow \{R^0_{j\, j=0,|R|}\}$ ▷ initial state
4: $E \leftarrow \emptyset$
5: **while** $i \leq |I|$ **do**
6: ISS.EXECUTE(I_i)
7: $RD \leftarrow$ ISS.READ(I_i) ▷ all resources read by I_i
8: $WR \leftarrow$ ISS.WRITE(I_i) ▷ all resources written by I_i
9: $V \leftarrow V \cup WR$ ▷ addition of new nodes
10: **for all** $R_w \in WR$ **do**
11: **for all** $R_r \in RD$ **do**
12: $E \leftarrow E \cup (R_w, R_r)$
13: **end for**
14: **end for**
15: $i \leftarrow i + 1$
16: **end while**

We define a resource dependency graph $G = (V, E)$ as follows. Let $\{R^l_i\}$ be a set of resources written by instruction I_l. Then the set of nodes V contains a node $\{R^l_i\}$ for each resource written by instructions belonging to I. Note that we assume that I_0 is an initialization instruction that updates all resources R. E is a set of ordered pairs (R^l_w, R^l_r), in which R^l_w and R^l_r are resources written and read by instruction I_l. We present a formal description of the dependency graph generation in Algorithm 1.

Algorithm 2 Dependency graph traversal

1: Input: $G = (V, E), C = \{R_{c1}, ..., R_{cn}\}$
2: **for all** $R_c \in C$ **do**
3: DFS(G, R_c)
4: **end for**
5:
6: **procedure** DFS(G, R_r)
7: VISIT(R_r)
8: $B \leftarrow \{R_b \exists (R_r, R_b) \in E\}$
9: **if** $B = \emptyset$ **then**
10: **return**
11: **end if**
12: **for all** $R_b \in B$ **do**
13: DFS(G, R_b)
14: **end for**
15: **end procedure**
16:
17: **procedure** VISIT(G, R_r)
18: **if** $mode = basic$ **then**
19: MARK(R_r)
20: **return**
21: **else**
22: **if** $mode = conditional$ **then**
23: $F \leftarrow \{R_f \exists (R_f, R_r) \in E\}$
▷ check if all output resources were cleared
24: **for all** $R_f \in F$ **do**
25: **if** $R_f \notin marked$ **then**
26: **return**
27: **end if**
28: **end for**
29: MARK(R_r)
30: **end if**
31: **end if**
32: **end procedure**

Let $C \subseteq R$ be a subset of resources of machine M. Algorithm 2 describes the creation of a program slice given a set of *faulty* resources C as an input. The same algorithm is also used for the justification heuristic. In that case the algorithm gets a set of *correct* resources C as its input.

The difference between the *basic* and *conditional* versions of the justification heuristic is embedded in the implementation

localized? (reduced slice size)	Number of scenarios	Program slice average size	Reduced slice average size
yes (1)	172	7.29	1
yes (>1)	91	9.34	3.00
no (0)	1	19.00	0
no (>0)	7	13.86	2.71

TABLE I: Experimental results - failure localization with *basic* justification heuristic

localized? (reduced slice size)	Number of scenarios	Program slice average size	Reduced slice average size
yes (1)	160	7.52	1
yes (>1)	108	8.87	3.16
no (0)	1	19.00	0
no (>0)	2	20.5	1.50

TABLE II: Experimental results - failure localization with *conditional* justification heuristic

of the VISIT function in Algorithm 2. During the traversal from a correct resource in the *conditional* heuristic, for each visited node we check if any remaining suspicious output resources exist that depend on it. The visited node will only be marked as cleared if it has no dependent suspicious output resources.

IV. RESULTS

We evaluated the effectiveness of our method by a series of experiments in which a single random error was injected into a test-case generated by the Threadmill exerciser [3]. We configured Threadmill to generate and run test-cases with about 220 random instructions. We executed these test-cases on a PowerPC ISS [8].

For each test-case, we created various error scenarios by selecting one instruction and altering its result. In each error scenario, we corrupted one modified resource after the instruction execution. This repeatedly caused Threadmill to stop and report the corrupted resources. The reports were then passed to our tool for analysis.

We generated a total of 271 error scenarios with a manifested data corruption at the final state. For each test-case, a dependency graph was created and dynamic program slicing was performed to identify the initial set of suspected instructions. Then, the two versions of justification heuristics were applied to reduce the slice size, and the final results were analyzed and compared. The tool was run on a single Intel Xeon Linux server operating at 2.4GHz with 16GB of RAM. The average run time of the entire analysis was less than one minute.

Our experimental results are reported in Table I and Table II. The data, divided into four categories, is presented in the order of the localization success. The first two categories represent successful localization of the failure. In the first category, the failure is precisely localized to a single-instruction set. In the second category, the faulty instruction is successfully localized into a limited set of suspicious instructions. The last two types represent unsuccessful localization cases. In one category, no suspicious instructions exist in the reduced set. In such a case, we expect the person driving the localization work to rely on the instructions in the program slice. In the last category, the reduced set contains some suspicious instructions, but none of those is the faulty one. Such a result is highly undesired, as it may mislead and increase the debug time.

As the results show, both heuristics provided highly efficient and accurate results. The *basic* heuristic provided precise failure localization in 97% of all cases, while in 63.5% of scenarios, the faulty instruction was the only instruction in the reduced set. The *conditional* heuristic successfully localized the failure in 99% of cases, while in 59% of scenarios, the heuristic accurately localized the suspicious instructions list to the single corrupted instruction. Although the average size of a reduced slice is smaller in the *basic* heuristic, a higher number of false results is observed when using this method. As explained in the previous section, this is due to the fact that the *basic* heuristic may lead to an erroneous clearance of faulty multi-output instructions in which only part of the outputs are corrupted. The reduced instruction set in the *basic* and *conditional* heuristics contained an average of 1.7 and 1.9 instructions, respectively.

In all the cases, the program slice, resulting from the execution of Algorithm 2, included the faulty instruction. This is attributed to the correct modeling of the architectural data propagation by the reference model.

In a single scenario, the final reduced instruction set ended up empty. In this case, the program slice before the justification phase contained 19 instructions. Finally, in two to seven scenarios, the suspicious instruction set was not empty, yet the faulty instruction was not contained in it.

A simple example of a faulty instruction that may be removed during the justification phase is in the case of the execution of a division instruction with a corrupted non-zero denominator, and a zero numerator. The result of such an instruction is 0, regardless of the numerator value and even though it used corrupted data. In such a case, nothing indicates that the instruction used a corrupted resource, and Algorithm 2 will continue to acquit preceding instructions (including the faulty one). This issue may be addressed in the future by having the ISS provide more accurate data regarding the dependency of the outputs on the inputs. For example in the example above, the output of the division instruction depends only on the numerator.

The detailed plots of sizes of initial and reduced slices are shown in Figure 5 and Figure 6 for all the analyzed failure scenarios. The graphs show the distribution of the slice sizes, as functions of the error location within the test-case. The results are divided according to the error origin memory or register. As shown, in all the cases the program slice is efficiently reduced to a small number of instructions (never exceeding 10 instructions). The overall efficiency of the method is particularly manifested in cases of big program slices, with up to 49 instructions. Gathering such a set of suspicious instructions would be a complex and time-consuming manual

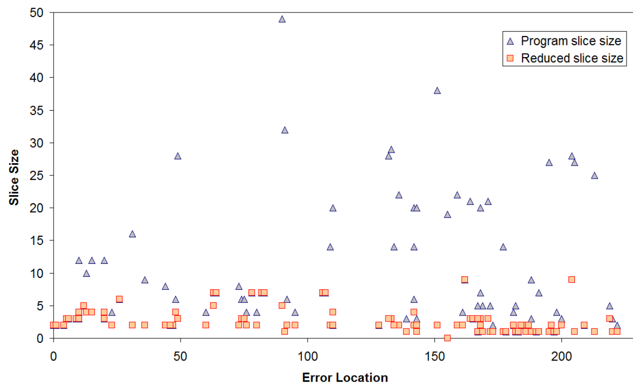

Fig. 5: Slice sizes as function of error location: errors appearing at registers

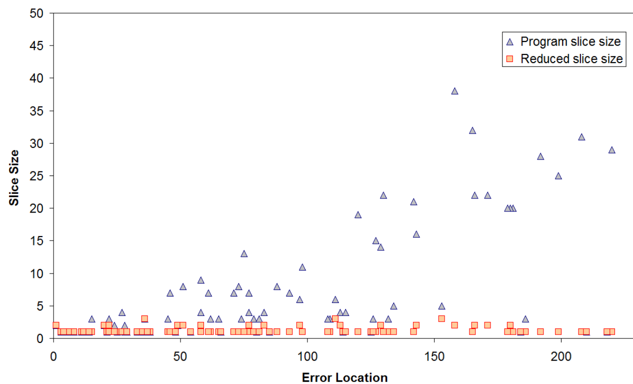

Fig. 6: Slice sizes as function of error location: errors appearing in memory

task and would significantly gain from the process automation. Moreover, all the program slices are successfully reduced to smaller sets by the justification heuristics, regardless of the original size of the program slice.

A specific trend can be observed in cases of error injection in memory resources (Figure 6). The plot shows a direct dependency between the error location and the size of the program slice. Errors injected in registers do not show such a trend. This effect can be explained by the fact that a relatively high number of memory resources are used by the test-cases as compared to the limited number of registers, as defined by the architecture. As a result, the chance of error propagation from a faulty memory resource to resources of *following* instructions is significantly lower than in the case of registers, regardless of the memory error location. On the other hand, any error appearing early in the test will have fewer *preceding* instructions that the slicing algorithm may mark as suspicious.

V. Conclusion and Future Work

We presented a novel method for the effective localization of post-silicon failures detected by consistency checking. Our method relies on the reference model of a design under test and utilizes dynamic program slicing. We also presented heuristics for the reduction of the set of suspicious instructions by leveraging the knowledge of the correct resources.

The experimental results demonstrate the effectiveness of our method. In over 97% of all cases, our method was able to narrow down the list of suspicious instructions to under 2 instructions, on average, out of over 200. In over 59% of all cases, our method correctly reduced a test-case to a single faulty instruction.

We plan to extend our method to assist in the localization of fails originating from multi-threaded test-cases and investigate heuristics that can explain control flow errors.

References

[1] ARM - fast model tools user guide. http://www.arm.com/products/tools/models/fast-models/index.php. Accessed: 2013-09-01.

[2] Abramovici et al. A reconfigurable design-for-debug infrastructure for socs. In *DAC*, pages 7–12, 2006.

[3] A. Adir et al. Threadmill: a post-silicon exerciser for multi-threaded processors. In *DAC*, pages 860–865, 2011.

[4] A. Adir, A. Nahir, and A. Ziv. Concurrent generation of concurrent programs for post-silicon validation. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 31(8):1297–1302, 2012.

[5] H. Agrawal, R. A. Demillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Software Practice and Experience*, 23:589–616, 1993.

[6] E. Anis and N. Nicolici. On using lossless compression of debug data in embedded logic analysis. In *ITC*, pages 1–10, 2007.

[7] R. Bloem et al. FoREnSiC- an automatic debugging environment for c programs. In *Haifa Verification Conference*, pages 260–265, 2012.

[8] P. Bohrer et al. Mambo: a full system simulator for the powerpc architecture. *SIGMETRICS Perform. Eval. Rev.*, 31(4):8–12, Mar. 2004.

[9] K. Chen, S. Malik, and P. Patra. Runtime validation of memory ordering using constraint graph checking. In *HPCA*, pages 415–426, 2008.

[10] K. Chen, S. Malik, and P. Patra. Runtime validation of memory ordering using constraint graph checking. In *HPCA*, pages 415–426, 2008.

[11] T. Corman, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. Cambridge MA. The MIT Press, 1990.

[12] F. M. de Paula, M. Gort, A. J. Hu, S. J. E. Wilton, and J. Yang. Backspace: Formal analysis for post-silicon debug. In *FMCAD*, pages 1–10, 2008.

[13] F. M. de Paula, A. Nahir, Z. Nevo, A. Orni, and A. J. Hu. Tab-backspace: unlimited-length trace buffers with zero additional on-chip overhead. In *DAC*, pages 411–416, 2011.

[14] A. DeOrio, J. Li, and V. Bertacco. Bridging pre- and post-silicon debugging with BiPeD. In *ICCAD*, pages 95–100, 2012.

[15] T. Hong et al. QED: Quick error detection tests for effective post-silicon validation. In *ITC*, pages 154–163, 2010.

[16] H. F. Ko and N. Nicolici. Automated trace signals identification and state restoration for improving observability in post-silicon validation. In *DATE*, DATE '08, pages 1298–1303, New York, NY, USA, 2008. ACM.

[17] D. Lin, T. Hong, F. Fallah, N. Hakim, and S. Mitra. Quick detection of difficult bugs for effective post-silicon validation. In *DAC*, pages 561–566, 2012.

[18] S.-B. Park and S. Mitra. Post-silicon bug localization for processors using IFRA. *Commun. ACM*, 53(2):106–113, Feb. 2010.

[19] Z. Poulos, Y.-S. Yang, and A. Veneris. A failure triage engine based on error trace signature extraction. In *On-Line Testing Symposium (IOLTS), 2013 IEEE 19th International*, 2013.

[20] S. Vasudevan, E. A. Emerson, and J. A. Abraham. Improved verification of hardware designs through antecedent conditioned slicing. *STTT*, 9(1):89–101, 2007.

[21] V. M. Vedula, W. J. Townsend, and J. A. Abraham. Program slicing for ATPG-based property checking. In *VLSI Design*, pages 591–596, 2004.

[22] I. Wagner and V. Bertacco. Reversi: Post-silicon validation system for modern microprocessors. In *ICCD*, pages 307–314, 2008.

[23] M. Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, July 1982.