社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2009－SLDM－139 (6)
2009－ＥＭＢ－ 12 (6)
2009／3／5

上位設計記述におけるダイナミックプログラムスライシングを用いたポストシリコンデバッグ支援手法

李 蓮福† 松本 剛史†† 藤田 昌宏††

† 東京大学大学院工学系研究科電機系工学専攻 〒 113–8656 東京都文京区本郷 7–3–1
†† 東京大学大規模集積システム設計教育研究センター 〒 113–0032 東京都文京区弥生 2–11–16
E-mail: lee@cad.t.u-tokyo.ac.jp, matsumoto@cad.t.u-tokyo.ac.jp, fujita@ee.t.u-tokyo.ac.jp

あらまし 機能的バグに対するポストシリコンデバッグは必須の作業になりつつあるが、チップレベルでは内部状態に対する controllability と observability が非常に低いなど、デバッグが大変である。本稿では、既存技術であるダイナミックプログラムスライシングと高位レベル設計手法を活用したバグ特定手法を提案する。提案手法では、まずデバッグ対象チップにおいてエラーが観測されたピンに該当する上位レベル設計記述上の変数に対するスライスを求める。同時に、スライスに含まれる各 statement が実行されたタイミングも共に記録する。その後、複数のエラーパタンから得られたスライスの共通集合を求めることにより、バグである可能性のある statement を求める。さらに、各スライスに含まれる statement の前後関係のパタンを比較し、共通するパタンを取り出し、正しいトレースとの差を求めることによって、バグである可能性を表す度合いに並べる。提案手法はその一部がツールとして実装されており、例題による予備的な実験結果を示す。
キーワード ハードウェアデバッグ, ポストシリコンデバッグ

A Post-silicon Debug Support Using Dynamic Program Slicing on High-level Design Description

Yeonbok LEE†, Takeshi MATSUMOTO††, and Masahiro FUJITA††

† Department of Electronics Engineering, School of Engineering, University of Tokyo,
Hongo 7–3–1, Bunkyo-ku, Tokyo, 113–8656
†† VLSI Design and Education Center, University of Tokyo,
Yayoi 2–11–16, Bunkyo-ku, Tokyo, 113–0032
E-mail: lee@cad.t.u-tokyo.ac.jp, matsumoto@cad.t.u-tokyo.ac.jp, fujita@ee.t.u-tokyo.ac.jp

Abstract Post-silicon debug of functional bugs is getting mandatory while it is a painful task due to the low controllability/observability of internal states of hardware chips. In this paper, we propose a bug localization method utilizing the existing concepts of dynamic program slicing and high-level design methodology. We derive slices for the output variable of the high-level design description corresponding to the output pin of target chip where an error trace is observed. At this time, we preserve the timing information on each statement of the slice as well. Then, assuming that multiple error patterns for the same bug are given, we extract the intersection of the slices for those error traces. Furthermore, we compare the execution patterns among the statements of each slice in terms of timing, then rank by means of the matching frequency which is regarded as suspicious degree to be the bug. We also developed a tool to achieve a part of our method and several experimental results are provided.
Key words HW debug, post-silicon debug

1. Introduction

First taped-out silicon chips fail due to various factors including not only manufacturing defects but also functional design bugs manifesting themselves as stopped response, wrong output and dropping packets. In spite of the effort

and progress on pre-silicon verification, it still cannot guarantee designs to be completely bug-free due to the limited development time. Thus, corner-case design bugs can escape to the chips. There is a study reporting that functional design bugs are responsible for 40 % of first silicon failures [1], which denotes that post-silicon debug is getting mandatory. Hereafter, this paper refers *bug* as the *functional design bugs*. Meanwhile, there is generally a great pressure of time-to-market for products where any inappropriate debug might cause extremely expensive results such as extra chip revisions, product delays or even product recalls.

In general, a debug process consists of 4 steps [2]; error detection, localization of the error, identification of the root cause, and correcting the bug, where there is lack of standardized methodology or automatic tool support. However, debugging post-silicon bugs by manual is generally a tedious and hard problem because of the following reasons. First, errors detected at post-silicon phase (i.e., errors that passed the pre-silicon verification) tend to originate from corner-case bugs and the error traces of them are typically very long which makes it difficult to analyze the root causes. What is worse, the observability and controllability of the internal states of chips are extremely low and strongly depend on the additional architectures prepared for debug such as scan chains or trace buffers. So far, several automatic methods for debug have been proposed. Some of these techniques target specific types of design errors [3] or specific architectures [4]. Others, which can deal with broader range, require the availability of a golden model to localize the bug [5] [6].

On the other hand, a number of researches have been conducted for debug automation for software system. Most of them are aiming to provide useful hints for localizing bugs. Among them, program slicing technique based approaches which focuses the dependencies inside the program [9] [10], and approaches focusing on the differences between the correct trace and error trace [11] have been the most popular.

On the basis of above situation, this work aims to alleviate the engineer's pain of chip debug with automatic bug localization. Our approach is largely motivated by the existing concepts; the high-level design methodology and dynamic program slicing [9].

High-level design methodology is to start the hardware design from higher abstraction levels (i.e., system level, behavior level) than traditional register transfer level (RTL). By working on the high abstraction levels, we can take advantage of comprehensibility of the design. Languages exist to describe the high-level design including not only Ansi-C but also C-like languages extended for hardware features such as SpecC [7] and SystemC [8] , and commercial tools for automatic synthesis of the design of those levels are available [13] [14]. Meanwhile, dynamic program slicing is a technique to extract a subset of program source code which consists of the portions that actually affect the value of the target variable at a certain point for a particular execution. The more detailed explanation of dynamic slicing is given in Section 2.

The main strategy of our method is to impose dynamic slicing onto the high abstraction level (i.e., behavior-level) design description with several error patterns each of which consists of a sequence of input/output events leading the chip to fail. With extracting the slices we collect the information of the executed timing of each slice element. After preserving the slicing results for all error patterns, we extract the intersection of those slicing results for the final result of the bug candidates.

The main contributions of this work are listed as follows.

- Proposed a method to localize the bug automatically
- Enhanced the efficiency of bug analysis by improved comprehensibility of design by adopting the high level description

The remainder of this report is organized as follows. Section 2. presents the basic knowledge of the dynamic slicing and its application to hardware design. Section 3. presents the detail of our proposed method for bug localization. Section 4. provides the experimental results applying our method follwed by the discussion of Section 5. Finally we summarize our work and present future works in Section 6.

2. Dynamic Program Slicing

Given a program, an input pattern and a target output variable, dynamic slicing identifies a subset of the executed program statements that *influenced* the target variable at an execution point. Its application range is very wide; from program debugging to complexity measurement. By the definition of *influencing*, several kinds of *slicing* exist as follows.

- *Data slicing*: Only data dependencies are traversed.
- *Full slicing*: In addition to data dependency, control dependencies are also traversed.
- *Relevant slicing*: In addition to full slicing, consider the data dependencies among the variables used in the path-condition predicates, and the data/control dependencies among those newly discovered portions.

When we apply the dynamic slicing, we have to decide which notion to employ considering the trade-off between the coverage to contain the actually influencing portions, the computational effort, and the number of the results as bug candidates. The relevant slicing provides the highest coverage to detect the bug but it contains too many redundant portions which are not actually related to the bug. A recent study [12] shows the results which demonstrates the fact that

full slicing is enough to reveal the bug using actual program examples containing bugs. Thus, we decide to adopt the notion of full slicing in this work.

We found several approaches to apply program slicing technique to hardware design. To deal with the event-driven characteristic of RTL language, Ichinose et al. introduced a new dependency, i.e., signal dependency [15]. Clarke et al. proposed a method to apply traditional program slicing as is on VHDL design description by converting the VHDL constructs onto the constructs for traditional procedural languages [16]. On the other hand, Tanabe et al. proposed an application of dynamic slicing onto the high-level hardware design language, i.e., SpecC [17].

3. Proposed Bug Localization Method

In this section, we present the detailed explanation of our proposed method to localize the suspicious portions to be the bugs utilizing dynamic program slicing.

3.1 Terminologies

Before going into the details of the proposed method, we define the terminologies used throughout this paper.

- An *error*: A functionally undesirable result such as stopped response, wrong output and dropping packets.
- A *bug*: A combination of the portions of a design which ultimately causes an error.
- A *pattern*: A pair of input/output sequences abstracted for high-level design. An *error pattern* stands for a pair of input sequence which lead the chip execution to fail and the resulting output sequence.
- A *trace*: A set of state instances with time stamp on an execution path. An *error trace* is a set of state instances with timestamp on an execution path of an error pattern.
- *timestamp*: A timing information when a statement was executed in terms of cycle.

3.2 A Bug Localization Flow

Figure 1 shows our bug localization flow which takes the following items as inputs.

- A high-level design description of the target design.
  - i.e.,A synthesizable behavior-level design described in SpecC [7]
- Multiple error patterns for the same bug
- A correct pattern the trace of which is similar to the intersection of multiple error slices.

For simplification of the problem, we also made several assumptions as follows.

- Only one bug is affecting the error at a time (a realistic and rationale assumption because a functional test vector is generally targeting single specific functionality [18]).
- Multiple error patterns are observed for the same error (i.e., same bug).
- The behavior level design description corresponds to the chip
- It is possible to take correspondence between the I/O ports of the chip and variables on the behavior-level description.
- Target error originates from the behavior level design itself.
- The behavior of the target design is deterministic.

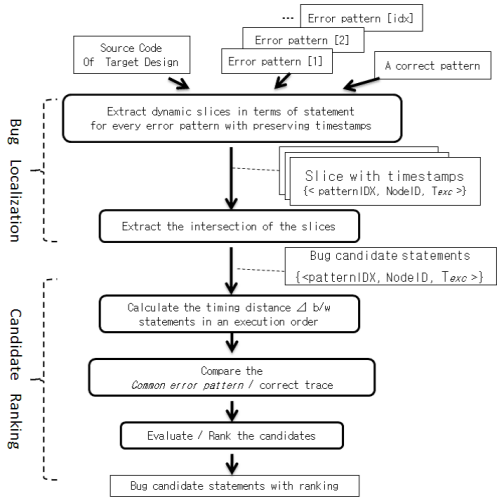

Figure 1 Our Bug Localization Flow

As shown in Figure 1, our bug localization method is mainly composed of two stages, bug localization and candidate ranking. Bug localization stage ultimately lists all the suspicious portions, while candidate ranking stage provides more efficiency by ranking the candidates by suspicious degree to be bug.

The detailed explanation of each step shown in Figure 1 is followed.

（1） **Bug Localization Stage**

- **Extract dynamic slices in terms of statement for every error pattern with preserving timestamps**

Given the inputs, we apply the dynamic program slicing on the behavior-level design description for the first step. While we evaluate the source code to compute the dynamic slices, we also collect timestamps for statements each of which represents the cycle when the statement is executed for further alanysis. The pseudo code of the proposed algorithm is presented in the Algorithm 1. In order to choose the dynamic execution path, we need to simulate the source code with concrete value assignment, i.e., *execute* each program statement. It is achieved by two procedures, *Execute* and *Eval*. As described in the Algorithm 1, *Execute* is performed for a statement existing in the source code in depth-first order.

An execution of a statement consists of evaluation of expressions involved in the statement and propagation of the concrete values assigned to the variables and decision of the next execution path. Updating and propagating slices for the variable are done during the *Eval* procedure. When the currently evaluating variable is a used variable to be referred by other variable in the same statement, the slice assigned to the variable until that point is preserved. If the currently evaluating variable is defined newly in the statement, the slice for the variable is updated with the current statement, current timing, and the preserved slices of used variables. Also, the slice which represent the control dependencies if there is any control dependencies to get to the current point is added. The timing information of the control slices are the latest executed time of them. After finishing the entire operation for an input pattern, we get a set of tuples of $< patternIDX, nodeID, T_{exc} >$ where $patternIDX$ is a number to distinguish the pattern, the $nodeID$ is a number to identify a statement, and T_{exc} is a timestamp for the statement of $nodeID$.

- **Extract the intersection of the slices**

Once the slices for all the input sequences of multiple error traces are obtained, the next step is to extract intersection of the slicing results obtained from the multiple error patterns. As for doing this, taking intersection is done by considering the commonly involved statements without considering timing information. We postulate that the bug resides in this intersection in this work and regard the intersection as the set of bug candidates. Thus, we convince that we can find the bug by just examining the statements of the candidate set. Our method is further useful to analyze the root cause than only computing the *slice* which is mere a set of portion of program, since we preserve the timing information (i.e., timestamp) for each statement. By exploring the behavior which causes the error the possibility to find the condition which the bug statement manifests itself (i.e., becomes erroneous value).

（2） **Candidate Ranking Stage**

For further improvement of debug efficiency, we are considering the following ideas in addition to the bug localization.

- **Caculate the timing distance Δ b/w statements in an execution order**

First of all, only for those nodes which are included in the intersection, we order the tuples of $< patternIDX, nodeID, T_{exc} >$ by T_{exc}, i.e., executed timing, for each error pattern. Then, we calculate the $\Delta_{ij} = (T_{exc}$ of $nodeID(j)) - (T_{exc}$ of $nodeID(i))$ between the statements. Using those data, we extract patterns which appear commonly in the error traces, which we call "*common error pattern*" (Figure 2(a)).

- **Compare the *common error pattern* / correct trace**

Now, we utilize the common intuition that a correct execution trace that is close to the error trace can provide a good hint to find the root cause of the error, which is originally presented in a theory of causality [20]. To explore the *close correct trace* (Figure 2(b)) and computing the difference between the correct trace and the common error pattern we can refer an existing research such as [11].

- **Evaluate / rank the candidates**

After getting all of the above disiderata, we rank the nodes of the intersection obtained from the first stage of bug localization by discent order of the score of difference (*distance* in [11]) with the correct trace.

Algorithm 1 Dynamic Full Slicing with timing Info.

```
Procedure Execute (s, crr_time, ctrl_depth)
if s is a predicate representing a condition then
  ++ctrl_depth;
  push(stack.time[s], crr_time);
end if
if s is a compound statement then
  for each sub_s do
    Execute(sub_s, curr_time, ctrl_depth)
  end for
end if
for each expression e in s do
  Eval(e, curr_time, ctrl_depth)
end for
end Procedure

Procedure Eval(e, crr_time, ctrl_depth)
for sub_e exists do
  Eval(sub_e, crr_time, ctrl_depth)
end for
rvalue_slice = {slice||slice =< s_id, {times} >};
ctrl_slice = {slice||slice =< s_id, {s_id, {times} >};
if ctrl_depth 0 then
  ctrl_slice = ctrl_slice ∪ {parent_ctrl_stmnts with latest executed time}
end if
if e is to be reffered then
  rvalue_dataslice = rvalue_dataslice ∪ e_i.dataslice ;
  rvalue_ctrlslice = rvalue_ctrlslice ∪ e_i.ctrlslice ;
else if e_iistobedefined then
  e.dataslice = e.dataslice ∪ {< s, times.insert(curr_time) > } ∪ rvalue_dataslice ;
  e.ctrlslice = e.ctrlslice ∪ rvalue_ctrlslice ;
end if
end Procedure
```

4. Experiments

To achieve the proposed bug localization method, we im-

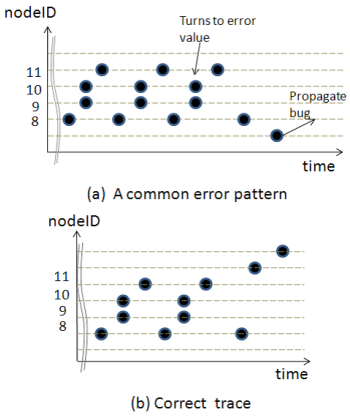

Figure 2 Comparation of Patterns

plemented the dynamic program slicing tool in C++. The tool can accept the behavior-level design source code written in SpecC. The tool is still under development to be able to handle more syntaxes and current scale of the entire source code is 17913 lines and. Addition of timestamp is not implemented in the tool.

To examine whether our method can detect the bug succesfully or not, we prepared several design examples written in SpecC language. A brief illustration of the example designs are as follows and each of which are containing a bug statement intentionally injected.

- **FACT** : Calculates a factorial of given natural number
- **IDCT1** (Inverse Discrete Cosine Transform) : Calculates 8 rows in 1 iteration
- **IDCT2** : Calculates 8 rows and 8 columns in 1 iteration
- **ELEV** : An elevator controller designed for 3-story building

The experimental results applying above examples to the first stage of our method which extracts the intersection of the multiple slices from the error patterns utilizing the dynamic slicing tool, are shown in the Table 1.

As shown in the experimental results presented in the Table 1, firstly we can see that the output result of applying our method could detect the intended bug succesfully in all cases. However, as for the 3 examples, FACT, IDCT1, IDCT2, we could not reduce the numbers of the bug candidates enough to distinguish the bug efficiently.

The main reason is because those programs are not essentially appropriate to take advantage of dynamic slicing. The advantage of dynamic program slicing is that we can focus on only for a particular execution path which is mostly related to the error using the concrete variable assignments. However, those 3 examples do not contain many variations of execution paths and most of the statements have data dependency among them. Thus, the more complicated examples which have a number of seperated modules (i.e., procedures) and execution paths are better to take the more advantage of dynamic program slicing. Meanwhile, the case of the 4th example, which has many execution paths, we could take considerable advantage of our method to reduce the number of bug candidates out of the entire statements while it could detect the intended bug as well.

5. Discussion

Still, there are a number of issues to be solved in order to improve the debug efficiency.

Above all, while the program slicing technique is effective to confine the program only to the related portions to target error, it is nessesary to support the hardware specific features different from the sequential programs. As mentioned in the Section 2. there are efforts to deal with the hardware features such as parallel behaviors. Thus this work extended the application to hardware debug assuming that they are already completed while actually they sloved the problem using a kind of trick on dependency graph, which are still not proved to be plausibility.

Also, there are many kinds of bugs which have various characteristics, and dynamic slicing are effective on just the syntactic bug which definitely resides only one bug, but not so effective for the bugs which manifesting themselves only when they combined with each other. On the other hand, debug with "distance metric" is strong for the bug statement which become erroneous value whenever it is executed, but week for the bug which become errorneous only under specific condition. For example, consider a statement "$x = y/(z\%200)$;" which becomes error only when the value of the z is multiples of 200. Thus it is desired to consider and apply the appropriate combination of the bug localization methods which can handle the various characteristics of real bugs.

Meanwhile, in order to apply the high-level information to the post-silicon chip debug, it is also required to take the correspondency between the high-level design elements and the components of the chip, as well as the behavior and data flow between those levels, while this work is performed under the assumption that the designer is able to take the correspondencies by manual.

6. Summary and Future Works

In this paper, focusing to post-silicon debug of the functional design bugs, we proposed a bug localization method utilizing the two existing concepts, high-level design method-

Table 1 Experimental Results

Design	LOC	# of Stmnts	# of branch	# of elements for the slice of error variable			# of elements of the intersection of slices for each pattern	the intersection includes buggy Stmnts?
				Error trace 1	Error trace 2	Error trace 3		
FACT	12	7	2	3	3	3	3	yes
IDCT1	44	39	0	19	19	19	19	yes
IDCT2	1084	923	8	616	616	616	616	yes
ELEV	324	239	122	7	7	7	7	yes

ology and dynamic program slicing technique. By imposing the dynamic program slicing on the high-level design description we can take advantage of high comprehensibility about the design which save the debug effort. Also, we proposed further improvement to enhance the bug efficiency by extracting the intersection of the error slices and ranking by the difference with the correct trace. To show the applicability and effectivity of our proposed method, we are developing a tool support to achieve the method. Some experimental results using the tool are introduced. There are still a number of remaining issues for further improvement of post-silicon debug.

For the future direction, we are planning to develope the bug candidate ranking algorithm as well as to propose a method that can handle more various kinds of bugs. For the next step to try, we are considering to make the candidate ranking method presented in section 3. to be a concrete algorithm examining existing pattern matching algorithms.

References

[1] P. Rashinkar, P. Paterson, and L. Singh, *System-on-a-chip Verification : Methodology and Techniques*, Kluwer Academic Publishers, 2002.

[2] D. Josephson and B. Gottlieb, "Silicon Debug", *Advances in Electronic Testing : Challenges and Methodologies*, pp. 77-108, 2006 Springer. Printed in the Netherlands.

[3] S-Y. Kuo, "Locating logic design errors via test generation and don't-care propagation", *Proc. European Design Automation Conference 1992*, pp. 466-471, 1992.

[4] S. Park and S. Mitra, "IFRA: Instruction Footprint Recording and Analysis for Post- Silicon Bug Localization in Processors", *Proc. Design Automation Conference 2008*, pp.373-378.

[5] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, "Fault Diagnosis and Logic Debugging Using Boolean Satisfiability", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 24, No.10, October 2005.

[6] Y. Yang, A. Veneris, N. Nicolici, "An automated software solution to silicon debug", *5th IEEE International Workshop on Silicon Debug and Diagnosis 2008*, pp.27-30, 2008.

[7] D. Gajski, J. Zhu, R. Doemer, A. Gerstlauer, and S.Zhao, *SpecC: Specification Language and Methodology*, Kluwer Academic Publisher, Mar. 2000.

[8] http://www.systemc.org/

[9] X. Zhang, H. He, N. Gupta, and R. Gupta, "Experimental Evaluation of Using Dynamic Slices for Fault Location", *Proc. AADE-BUG 2005*, pp.33-42, 2005.

[10] J. R. Lyle and M. Weiser, "Automatic program bug location by program slicing", *In 2nd International Conference on computers and Applications 1987*, pp.877-882, 1987.

[11] A. Groce, S. Chaki, D. Kroening, and O. Strichman, "Error explanation with distance metrics", *International Journal on Software Tools for Technology Transfer*, Vol.8, Issue 3, pp.229-247, June 2006.

[12] X. Zhang, N. Gupta, and R. Gupta, "A study of effectiveness of dynamic slicing in locating real faults", *Empirical Software Engineering*, Vol. 12, No. 2, April 2007, pp. 143-160.

[13] Catapult C Synthesis: http://www.mentor.com

[14] PICO Express : http://www.synfora.com/

[15] S. Ichinose, M. Iwaihara, and H. Yasuura, "Program slicing on VHDL descriptions and its evaluation", *IEICE Trans. Fundamentals*, Vol. E81-A, No.12, pp. 2585-2594, Decenber 1998.

[16] E. M. Clarke, M. Fujita, S. P. Rajan, T. Reps, S. Shankar, and T. Teitelbaum, "Program Slicing of Hardware Description Languages", *in Proc. 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pp.298-312, 1999.

[17] K. Tanabe, S. Sasaki, M. Fujita, "Program Slicing for System Level Designs in SpecC," *Proc. of IASTED International Conference on Advances in Computer Science and Technology*, pp.252-258, Nov. 2004.

[18] L. Huisman, "Diagnosing arbitrary defects in logic designs using single location at a time (SLAT)," *IEEE Trans. on CAD*, vol.23, no.1, pp.91-101, Jan.2004.

[19] K-H Chang, V. Bertacco, and I. L. Markov, "Simulation-Based Bug Trace Minimization With BMC-Based Refinement", *IEEE Transactions on Computer-Aided Design of Integrated Circuits And Systems*, Vol.26, No.1, January 2007.

[20] D. Lewis, "Causation", *Journal of Philosophy*, 70, pp.556-567, 1973.