Microprocessors and Microsystems 000 (2016) 1–13

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Debugging hardware designs using dynamic dependency graphs

Jan Malburg[a,b,1,*], Alexander Finder[a,2], Görschwin Fey[a,b]

[a] University of Bremen, Bibliothekstr. 1 (MZH), 28359 Bremen, Germany
[b] German Aerospace Center, Robert-Hooke-Str. 7, 28359 Bremen, Germany

ARTICLE INFO

Article history:
Received 6 August 2015
Revised 20 May 2016
Accepted 19 October 2016
Available online xxx

Keywords:
Dynamic dependency graphs
RTL
Fault localization
Debugging

ABSTRACT

Debugging is a time consuming task in hardware design. In this paper a new debugging approach based on the analysis of dynamic dependency graphs is presented. Powerful techniques for software debugging, including reverse debugging, dynamic forward and backward slicing, and spectrum-based fault localization are combined and adapted for hardware designs. A case study on designs with multiple faults approved the power of the proposed debugging methodology reducing the debugging time to 50% in comparison to conventional techniques.

1. Introduction

Consuming more than 46% of the total ASIC development effort, verification already is a bottleneck in the design flow [1]. The fastest growing component of the verification process, and with 60% already the largest one, is debugging, i.e. fault localization and error correction. To a large extent debugging is still a manual task, where a developer must understand an error based on a sequence of input values for which the design yields incorrect outputs. Afterwards, he has to localize and fix the fault in the design. This is normally done using a simulator which presents the values of the different variables in a waveform.

Several techniques for helping the developer in debugging have been proposed. Reverse debugging has already been applied on software, reducing the debugging time to 25% [2] and is, for example, implemented in the commonly used software debugger GDB [3]. Dynamic forward and backward slicing reduces the amount of code the developer has to inspect in order to find and fix a bug [4]. Spectrum-based fault localization uses program spectra of correct and incorrect simulation runs to identify parts of the code which are likely to contain a bug [5]. This gives the developer further guidance while searching for the fault that causes the error.

In this paper we present the following debugging techniques for *Hardware Description Languages* (HDL):

- The use of reverse debugging
- Dynamic program slicing which considers control dependency
- Spectrum-based fault localization based on dynamic dependency graphs

For the implementation of our approach we use dynamic dependency graphs [6], which we additionally annotated with the values of the operands for each vertex of the graph. We implemented a *Graphical User Interface* (GUI) allowing the developer to investigate single simulation runs of his design. To improve the developer's understanding of a run, the GUI presents the run as a graph with the executed source code as vertices, dependencies as edges, and temporal behavior as time line. The graph can be shown in different granularities. In addition, the user is able to combine different granularity-levels with each other.

We conducted a case study on designs with multiple faults in which we compared our approach with a conventional debugging approach. In this case study, our approach showed a reduction of debugging time by 50% compared to the conventional approach.

The remainder of this paper is organized as follows: In Section 2 we summarize related work. Section 3 presents definitions used throughout the paper. In Section 4 our approach is introduced in detail. The evaluation of our approach is given in Section 5 and Section 6 concludes the paper.

2. Related work

So far, there exist several methods to support a developer in debugging a design both for software [2,4,5,7–10] and for hardware [11–16].

* Corresponding author.
E-mail addresses: Jan.Malburg@dlr.de (J. Malburg), alexander.finder@daimler.com (A. Finder), Goerschwin.Fey@dlr.de (G. Fey).

[1] During the work Jan Malburg changed to German Aerospace Center.

[2] Present affiliation: Daimler AG, Hanns-Klemm-Str. 45, 71034 Böblingen, Germany

http://dx.doi.org/10.1016/j.micpro.2016.10.004
0141-9331/

A typical approach for software is to reduce the amount of code a developer has to inspect in order to find a fault. One of these techniques is static program slicing, proposed by Weiser [7]. For a given set of program points (slicing criterion), static program slicing computes all statements, for which an input to the system exists, such that those statements influence the slicing criterion (backward slicing) or can be affected by the slicing criterion (forward slicing).

While debugging, a developer normally knows an execution for the system which reveals an error. Korel and Laski [4] proposed dynamic program slicing which computes the statements that affect (or are affected by) the slicing criterion under a given input. Zhang et al. [17] describe three different types of dynamic slicing. In *data slicing* only data dependencies are included. For *full slicing* control dependencies are additionally contained and for *relevant slicing* also statements are considered which could change the value of a variable by changing the system's execution path.

Another approach to reduce the amount of code a developer must investigate, is automated fault localization which computes parts of the source code that might contain a fault. In spectrum-based fault localization, program spectra, for example, coverage information of failing and succeeding test cases, are used to localize a fault. A well-known tool applying spectrum-based fault localization is the visualization tool Tarantula [5]. The tool colors statements in the source code depending on their suspiciousness of causing an error. The suspiciousness is computed by comparing the amount of failing and succeeding runs which execute the statements.

A technique closely related to spectrum-based fault localization is spectrum-based feature localization [18]. Both techniques use coverage information and heuristics to compute whether some part of the code is related to some behavior of the system. In case of fault localization this is the cause of an incorrect behavior and in case of feature localization the code responsible for an intended behavior. The heuristics used for spectrum-based fault localization can, with only small changes, be applied to spectrum-based feature localization [19].

In [8] Renieres and Reiss present another technique for fault localization. Their approach searches for a successful test case with minimal difference to the failing test case. Then those parts of the code are reported as fault candidates which are covered by the failing run, but not by the successful run. Groce et al. [9] further formalized the approach using a model checker to generate a successful run with minimal distance to the failing run.

Delta Debugging [10] aims at helping a developer in finding a fault by isolating the possible trigger of the fault. Delta Debugging originally was presented to determine those changes to a software system which cause a regression error. Later Delta Debugging has been extended to minimize error revealing inputs to the system and to find the minimal change which must be applied to a failing run such that the result becomes correct [20]. The intuition is that knowing the trigger of an error helps in understanding the error and a smaller test input executes less code of the system such that the possible fault locations can be restricted.

In [2] Lewis describes a tool for Java programs called *Omniscient Debugger* which allows to step backwards in the execution of a program. In this approach, first an instrumented version of the program is executed and all events, i.e. assignments to variables, function calls, returns from method calls, etc. are stored. Then a GUI answers questions in the form of "Where has the value of this variable been set?" and allows the user to step forward and backward through the program's execution.

Clarke et al. [11] developed an adaption of static program slicing for code in HDL. In their approach they relate HDL-constructs to constructs of software languages.

Path tracing [12] follows the controlling inputs of different gates to determine the critical path which is responsible for the value of a given signal. In [13] a description is given how to apply path tracing on HDL-level. However, the approach only considers data dependency and neglects control dependency. Also all kinds of path tracing neglect forward dependencies. Altogether, path tracing can be understood as backward dynamic program slicing applied to hardware systems.

In [21] Stumptner and Wotawa describe model-based diagnosis [22] for hardware description languages. In their approach they generate different versions of the design based on a set of construction rules to decide whether the incorrect behavior can be fixed by this change, i.e., the inverse of the change explains the incorrect behavior. They use the number of changes which must be applied to the design as a ranking function for their explanations such that fewer changes are considered better explanations. Often program slicing is applied for model-based diagnosis to reduce the amount of possible changes which has to be considered [23].

Another type of model-based diagnosis for hardware designs is *Satisfiability* (SAT)-based debugging [14]. Given a set of stimuli to a design which result in the violation of the specification, SAT-based debugging uses a SAT-solver to explain the incorrect behavior of the design. The circuit is transformed into a Boolean formula and the SAT-solver determines a minimal number of fault candidates which must be corrected to fulfill the specification. However, the SAT-based approach is limited to the capability of the underlying SAT-solver making the application unfeasible for larger designs.

In [15] two analysis, "What-if" and "How-can" are presented which are based on dynamic data flow analysis. "What-if" analysis computes how the change of one or several values affects a target value. The idea is similar to a conventional debugger, where a user can change the value of the variable during debugging. "How-can" analysis computes the values for a set of signals, such that a target signal gets a desired value. This analysis is similar to the idea of SAT-based debugging. To prevent high runtimes they limit the set on which the search is conducted to 70 binary variables, what however, also limits the practical benefit of the analysis.

Beer et al. [16] describe a technique which computes the cause of the violation of a specification given as a *Linear Time Logic* (LTL)-formula. The technique requires the specification and a violating simulation run. However, they do not consider expressions or statements as cause for a violation, instead they consider the valuation of one or more variables as the cause of the violation. Hence their approach does not give any information why the valuation is wrong at this point. The reason could be that an assignment is missing or that the previous assignment was wrong. Further, their approach does not distinguish between the case that changing the variable found would fix the violation and the case that changing the variable found only pushes the violation to another clock cycle.

In [24] Le, Große, and Drechsler present a technique for SystemC *Transaction-Level Modeling* (TLM) fault localization. Their approach targets faults at the level of transaction, synchronization, and timing. Thus, they do not consider faults in the form of incorrect assignment to variables, incorrect formulas or similar. If a design is erroneous, they create a set of alternative designs based on their fault model. These changes to the design are parametrized. Then a model checker is used to check whether a changed design is fault free for some concrete parameter value. Those designs which can correct the fault are reported as diagnosis of the error. Compared to our approach they target different kinds of faults. First of all, they target typical faults introduced at the TLM-level, for example synchronization errors between transactions. These are not common faults at the RT-level at which our approach works; in fact, transactions like in TLM do not exist at the RT-level.

Please cite this article as: J. Malburg et al., Debugging hardware designs using dynamic dependency graphs, Microprocessors and Microsystems (2016), http://dx.doi.org/10.1016/j.micpro.2016.10.004