A Cache Based Algorithm to Predict HDL Modules Faults

José Augusto M. Nacif[†,§] Thiago S. F. Silva[†], Luiz Filipe M. Vieira[†],
Alex Borges Vieira[‡], Antônio O. Fernandes[†], Claudionor N. Coelho Jr.[†]
[†]Universidade Federal de Minas Gerais, Belo Horizonte, Brazil
[§]Universidade Federal de Viçosa, Florestal, Brazil
[‡]Universidade Federal de Juiz de Fora, Juiz de Fora, Brazil
E-mail:{jnacif,thiagosf,lfvieira,borges,otavio,coelho}@dcc.ufmg.br

Abstract—**Verification is the most challenging and time consuming stage in the integrated circuit development cycle. As designs complexities double every two years, novel verification methodologies are needed. We propose an algorithm that dynamically builds and updates an HDL module error proneness list. This list can be used to assist the development team to allocate resources during verification stage. The algorithm is build up using the idea that problematic modules usually hide many uncovered errors. Thus, our algorithm caches the most frequently modified and fixed modules. In an academic experiment composed by 17 modules, using a cache of size 3, we were able to correctly predict almost 80% of the faults occurrences.**

I. Introduction

Verification of modern integrated circuits demands most part of the development efforts [1]. As complexity increases drastically, new strategies to improve design verification are needed. Considering that nowadays hardware and software development cycles have many aspects in common, there are software engineering techniques that can potentially improve integrated circuit verification. Software evolution research is focused in studying the process of developing and updating software systems. During the software development cycle, this information is recorded by Version Control System (VCS) and Bug Tracking System (BTS) tools. In this field there are many works that use design complexity and revision history information to build software subsystems fault proneness prediction models.

Predicting HDL modules fault proneness can be also useful in integrated circuit design cycles. During commercial integrated circuits development, tight time constraints are commonly imposed by market forces. Thus, the manager must allocate the verification resources (i.e. verification engineers and CPU time) wisely. In order to assist the manager in this task we propose the use of a dynamically build HDL module error proneness list. Our main contribution is to use revision history information to build this list. The algorithm used to predict the most fault prone modules is based on the concept of cache.

This paper is outlined as follows. Section II reviews similar techniques to predict software subsystems fault proneness. In Section III, we present our algorithm to predict HDL modules fault proneness while in Section IV results of an academic experiment are discussed. Finally, in Section V, we conclude our remarks and present future work directions.

II. Related Work

Revision history information mining is an active research area in software engineering [2]–[5]. There are many works that take advantage of this information to estimate development effort, impact of changes in source code and defect prediction. Despite the similarities between software and integrated circuits design, the practice of retrieving design history information is still incipient on hardware community [6], [7]. Thus, we present the most relevant software engineering works that use similar approaches.

Hassan and Holt [8] propose a cache inspired strategy to dynamically predict fault occurrences. They build a list of ten most fault prone software subsystems. Each subsystem is equivalent to a source code subdirectory. Six large open-source softwares are used as raw data. The development history is retrieved from a VCS and classified as fault repairing modifications, general maintenance modifications or feature introduction modifications. The list replacement algorithm uses the following strategies: most frequently modified, most recently modified, most frequently fixed, and most recently fixed. In order to compare the results, an adjusted hit rate and an average prediction age are also calculated and presented.

In [9], Kim *et al.* observe that bug occurrences are concentrated in four localities:

1) **Changed-entity:** The most recently changed code is more likely to contain new bugs;
2) **New-entity:** New entities included in the code are usually more prone to bug occurrences;
3) **Temporal:** The idea of temporal locality is that faults are inserted by developers when they are fixing the code;
4) **Spatial:** The spatial locality is related with logical coupling among entities. In order to identify coupling, revisions transactions are retrieved and files are grouped by commits with the same motivation.

Kim *et al.* [10] also investigates a methodology that collects software code parts and classifies them using keywords. The revision number is used to correlate modifications reported in a BTS tool with VCS repository transactions, making possible the keyword classification. Instead of pointing which modules are error-prone, they list source code parts that are prone to contain uncovered bugs. As design evolves, the predictor efficiency is evaluated using the hit rate metric.

III. Prediction Strategy

During development process, the design team implements integrated circuit modules in HDL populating the VCS repository. Thus, the team members have access to the design revisions and bug database. It is common in industry to divide the team in two groups:

1) Design engineers: responsible for the integrated circuit design and implementation. This team also fixes reported errors;
2) Verification engineers: responsible for validating the correct behavior of the design using simulation or formal tools. When this team identifies an incorrect behavior a bug report is issued.

The idea behind our prediction strategy is that modules that have many reported errors or modifications tend to concentrate the undetected bugs. In order to dynamically capture the bug and modification history of the design, our prediction algorithm accesses both, VCS Repository and Bug database, as shown in Figure 1. The predictor dynamically generates a list of the most fault prone modules.

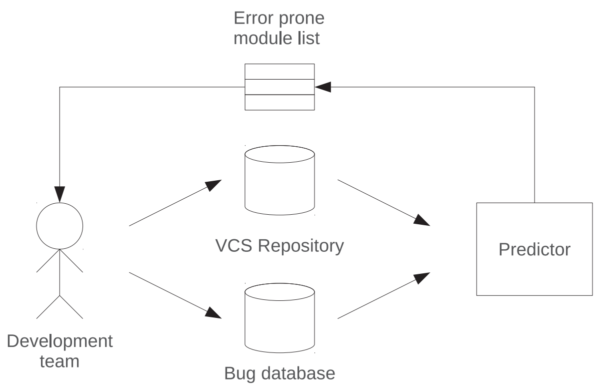

Fig. 1. Context in which the error proneness prediction algorithm is used.

The prediction algorithm is presented in Figure 2. The error-prone module predictor iterates over a list of modules holding on its cache the modules that satisfy a given replacement policy. The replacement criteria can be used to create different lists, making possible to the predictor to consider different aspects of temporal locality such as general change history or bug history.

IV. Experimental Results

In order to evaluate the effectiveness of our algorithm, we have collected data from an academic project over four months. Student teams developed a simplified MIPS32 architecture implementation with 17 modules. All project data is extracted from a Subversion (SVN) [1] repository and from a

[1]SVN is a popular VCS tool that is freely available

```
1:  $initialize\ Cache$
2:  for $i \leftarrow 1$ to $|SelectedRevisionsModulesList|$ do
3:      $module \leftarrow SelectedRevisionsModulesList[i]$
4:      if $module\ is\ not\ already\ on\ the\ cache$ then
5:          if $|Cache| = CACHE\_SIZE$ then
6:              $replace\ old\ module$
7:          else
8:              $load\ new\ module$
9:          end if
10:     end if
11: end for
12: for $i \leftarrow 1$ to $|Cache|$ do
13:     print $“Error\ prone\ module\ \#” + i + “ : ” + Cache[i]$
14: end for
```

Fig. 2. Fault proneness prediction algorithm.

bug database. Our bug report database is populated using a language [11] that facilitates automatic extraction. All project commits are classified in two categories:

1) Bug fix: these commits are motivated by a bug fix. The total number of commits that fall in this category is 47;
2) Design resume: when a designer modifies the source code. There are 159 design resume revisions.

Our predictor is designed with three different cache replacement strategies: most frequently fixed (MFF), most frequently modified (MFM), and first in, first out (FIFO). MFF uses bug fix commits and MFM design resume commits. The cache size is arbitrated in sizes 3 and 5 which represents 23.08% (3/13) and 38.46% (5/13) of 13 modules found with errors, and 17.65% (3/17) and 29.41% (5/17) of 17 modules modified during design stage. In order to have a comparative parameter we introduce random hit probability. This number is calculated according to the cache size, and total number of modules.

Figures 3 and 4 depict the hit rate results of our algorithm using MFM and FIFO replacement policy with cache sizes 3 and 5, respectively. In both Figures, during the first revisions, while the number of known modules is smaller than cache size, the hit probability tends to correctly predict 100% of the fault occurrences. As the number of known modules increases, the number of hit probability algorithm decreases, and both MFM and FIFO algorithms reach hits rates around 80%. The MFM replacement policy performs better than FIFO. Using a cache of size 5 slightly improve the performance of both algorithms.

Figures 5 and 6 present the prediction algorithm results using MFF with a cache of sizes 3 and 5, respectively. It is important to note that for these results we considered only modules that had reported faults (13, out of 17). The MFM results perform better than MFF, which reaches a hit ratio of around 55%.

V. Conclusion and future work

We proposed a simple error-prone prediction algorithm to be used in HDL modules. The proposed algorithm uses integrated circuit revision history information to include the most error-prone modules in a list that is similar to the cache concept.

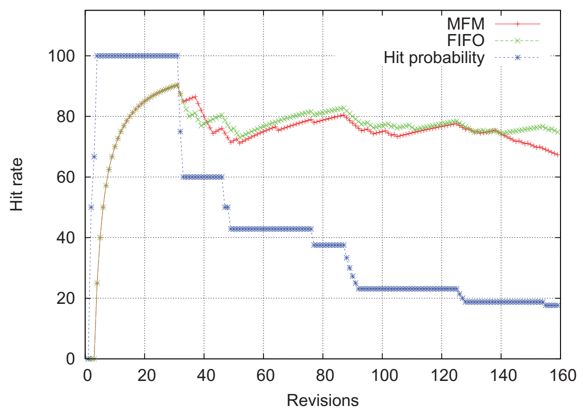

Fig. 3. Fault proneness prediction using cache size 3 (MFM).

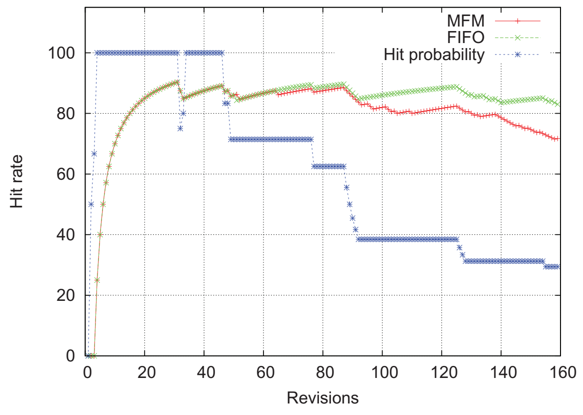

Fig. 4. Fault proneness prediction using cache size 5 (MFM).

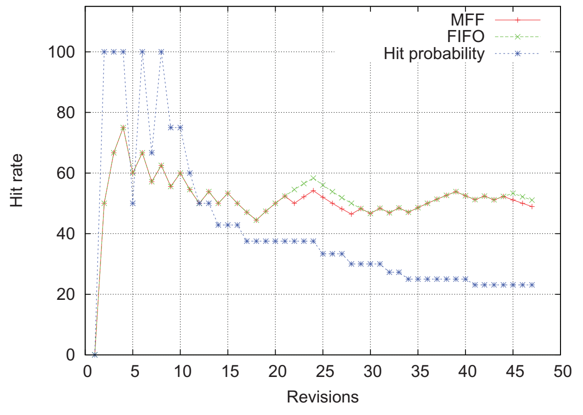

Fig. 5. Fault proneness prediction using cache size 3 (MFF).

In our results we showed that the most effective replacement policy is to keep in the list the most frequently modified modules. Using this strategy, we were able to predict around 80% of the fault occurrences. This replacement strategy performed better because modules that suffer frequent modifications tend to contain more uncovered faults.

Future work includes further investigation of new replacement algorithms, and new open source projects. We also intent to explore the correlation among spatial metrics and faults.

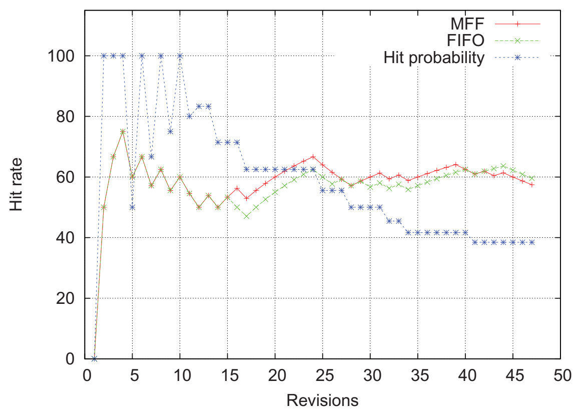

Fig. 6. Fault proneness prediction using cache size 5 (MFF).

ACKNOWLEDGEMENTS

This work is supported by CNPq and FAPEMIG under grants #PNM-141201/2005-3, #CEX-1485-06, and #APQ-02217-10.

REFERENCES

[1] "International Technology Roadmap for Semiconductors," Available at: http://www.itrs.net, 2009.

[2] M. Fischer, M. Pinzger, and H. Gall, "Analyzing and relating bug report data for feature tracking," in *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2003, p. 90.

[3] D. German, A. Hindle, and N. Jordan, "Visualizing the evolution of software using softChange," *International Journal of Software Engineering and Knowledge Engineering*, vol. 16, no. 1, pp. 5–21, 2006.

[4] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey, "Facilitating software evolution research with kenyon," in *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA: ACM, 2005, pp. 177–186.

[5] G. Gousios and D. Spinellis, "Alitheia core: An extensible software quality monitoring platform," *Software Engineering, International Conference on*, vol. 0, pp. 579–582, 2009.

[6] J. A. Nacif, T. Silva, A. Tavares, A. Fernandes, and C. Coelho, "Efficient allocation of verification resources using revision history information," in *Design and Diagnostics of Electronic Circuits and Systems, 2008. DDECS 2008. 11th IEEE Workshop on*. IEEE, 2008, pp. 1–5.

[7] J. Nacif, T. S. F. Silva, L. F. M. Vieira, A. B. Vieira, A. O. Fernandes, and C. N. Coelho, "Tracking Hardware Evolution," in *International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2011.

[8] A. Hassan and R. Holt, "The top ten list: Dynamic fault prediction," in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*. IEEE, 2005, pp. 263–272.

[9] S. Kim, T. Zimmermann, E. Whitehead Jr, and A. Zeller, "Predicting faults from cached history," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 489–498.

[10] S. Kim, K. Pan, and E. E. J. Whitehead, Jr., "Memories of bug fixes," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. SIGSOFT '06/FSE-14. New York, NY, USA: ACM, 2006, pp. 35–45. [Online]. Available: http://doi.acm.org/10.1145/1181775.1181781

[11] T. Cardoso, J. Nacif, A. Fernandes, and C. Coelho, "BugTracer: A system for integrated circuit development tracking and statistics retrieval," in *Test Workshop, 2009. LATW'09. 10th Latin American*. IEEE, 2009, pp. 1–4.