Software Defect Prediction for LSI Designs

Matthieu Parizy, Koichiro Takayama, Yuji Kanazawa
Design Engineering Lab.
FUJITSU LABORATORIES LTD.
Kawasaki, Japan
{parizy.matthieu, k.takayama, ykanazawa}@jp.fujitsu.com

Abstract—While mining software repositories is a field which has greatly grown over the last ten years, Large Scale Integrated circuit (LSI) design repository mining has yet to reach the momentum of software’s. We felt that it represents untouched potential especially for defect prediction. In an LSI, referred to as hardware later on, verification has a high cost compared to design. After studying existing software defect prediction techniques based on repository mining, we decided to adapt some for hardware design repositories in the hope of saving precious resources by focusing design and verification effort on the most defect prone parts of the design. By focusing our resources on the previously mentioned parts, we hope to improve our designs quality. We discuss how we applied these prediction techniques to hardware and show our results are promising for the future of hardware repository mining. Our results allowed us to estimate a possible total verification time reduction of 12%.

Keywords—defect prediction; hardware; LSI; repository mining; code change; code metrics;

I. INTRODUCTION

Since the 1980’s advent of Electronic Design Automation, hardware is designed using Hardware Description Languages (HDL). These languages present similar features to software programming languages in structure and in syntax such that it has lead hardware design to reach higher level of abstractions. This had the impact that in many aspects HDL code is handled the same way as software, such as being parsed to produce code with a lower level of abstraction or, in the end, integrated circuit layout data. HDL development relies also on the same Version Control Systems (VCS) and this is what makes software repository mining techniques usable for hardware as well.

On the verification side, hardware verification is becoming more and more complex for design as scale of chips is exploding reaching several billions of transistors. One of the major differences between software and hardware verification is that on the hardware side, everything behaves in parallel in a chip compared to software which still relies deeply on sequential behavior.

As verification cost occupies a large part of the total design budget, usually more than 70%, allocating verification budget properly is crucial. One aspect which is different from software verification is that, for example, if you design a chip that will run at 1GHz, you verify it first using simulation which can simulate the chip’s behavior with the design internal clock running at only a few kHz. To verify the chip’s behavior and have the design clock running at a speed of several MHz, you use hardware accelerated emulators which can imitate the chip’s behavior from its HDL description. The problem is they are extremely costly reaching the order of the millions of dollar per year for run cost. Therefore, it is crucial for the verification engineers to focus on the most defect prone parts of the design i.e. which have the lowest quality to lower the cost of verification.

We make the hypothesis that defect prediction can help both hardware verification and design. It can help source code review process by focusing more on defect prone parts of the design. It can help verification planning by giving highest verification priority to the same parts. The place where we think defect prediction would fit in hardware design and verification flow is shown in Fig. 1.

Fig. 1 represents both verification and design tasks as well as their interaction with the project’s repository. On the right side, designers write and implement specifications, commit their source code on the repository after code review, fix defects found by verification engineers. On the left side, verification engineers make a test plan from analyzing the design’s specifications, they then implement the tests according to the plan and report any defect found to the design team.

Fig. 1. Flow of defect prediction in hardware design and verification

On a large scale design, there are several ways to find the defect prone parts. You can rely on past defects, coverage data to this end but sometimes it is not enough as some parts of the design can be highly covered and have few defects in the past.
history but could still hide some defects. High coverage, low defect found rate would traditionally lead us to set the verification priority of such parts to “low”. That is why we need a method allowing us to predict defects so we do not lower the priority of such parts.

Software Repository Mining has greatly matured over the last decade. This maturity can be embodied by work such as the MSR Cookbook[1] which provided us with reliable hints on how to proceed for hardware repository mining, such as being aware of the complexity of text mining. [2].

By studying the work of D’Ambros, Lanza and Robes [2], which compares several defect prediction approaches extensively, we assessed the defect prediction method relying on entropy of code changes[3] would be a fit for hardware defect prediction. The reason being that since static analysis tools for HDLs are unpopular and immature, the set of metrics extracted from HDL code is insufficient and [3] relies solely on the activity of source code revision control system. Therefore it could be implemented on hardware repositories “as is”.

[2] also provided us efficient ways of evaluating the predictive power of defect prediction models on hardware. To compare our results with [2] we relied mainly on the calculation of the Spearman Correlation coefficient between our model and our actual defect data.

Our contribution is that we successfully adapted the above mentioned works on a hardware design. Adapting this technology allowed us to estimate a possible total verification time saving of 12%.

This paper is organized as follows: In section 2 we discuss related works. Section 3 describes our hardware defect prediction model. Section 4 presents the results of our case study. Finally, in Section 5 we summarize our contributions and discuss our future work.

II. RELATED WORK

To the best of our knowledge, there are only few works on mining hardware repositories compared to software. One of the work which caught our interest was the work of Nacif et al.[4] on tracking hardware evolution. They defined and then tracked several metrics, such as flip-flops number, logic gates numbers, over time on hardware projects in a framework they created. The purpose of their framework is to improve design quality. Inside their framework, they evaluate error proneness with an algorithm which is based on the idea that most frequently fixed modules are the most defect prone whereas we focus on commit patterns.

As mentioned in the introduction, [2] provided us a panel of software defect prediction techniques as well as how to benchmark them. What we judged was the most promising to adapt on hardware was [3], the entropy of code changes (ECC), as well as D’Ambros own entropy of code metrics (ECM) which replaces the code changes by various code metrics, such as classes FanIn and number of attributes, as input of the entropy model from Hassan[3].

Our reasoning behind our decision was that:

- Entropy of source code metrics changes showed one of the best results in [2].
- Hardware architecture presents similarities with object-oriented software designs. Hardware modules, the basic units of hierarchy in HDL, are the building blocks of the chip. A module can be instantiated inside another module. The difference with software is when a module is instantiated, it represents a physical part of the chip. Therefore instantiated modules could be compared to classes in software, instantiated at the start of a program, which are never deleted and have all their methods “running” in parallel as long as the circuit is “on”. Most of modules’ input come from other modules’ output and similarly most of modules’ output are used as inputs of other modules. This leads to dependency problems being similar to software’s.
- The above described similarities led us to assume that hardware description changes must happen in a similar fashion to software code changes. Therefore adapting the entropy of code changes to hardware should be straightforward.
- If the entropy of hardware description changes shows promising results, then we could also replace description changes by hardware metrics and hopefully obtain even better results the same way D’Ambros did for software. For example we used modules’ number of LOC changed for each time period. In future work we will also try giving a different weight to changes due to defect fixes, compared to changes due to specification extensions.

Fig. 2 illustrates the activity of code changes on a Version Control System (VCS) for four different files, which represent hardware modules in our case. These files changes are committed to the VCS several times on three periods of, for example, two weeks. ECC model is based on Shannon entropy in order to attribute a higher risk to changes spread over multiple files rather, like period 2 in Fig. 2, than on changes focusing on a single file, like in period 3. This difference in risk is illustrated by the difference in size of the circles on Fig 2. The rationale is that changes spreading over multiple files is much more complex, thus more defect prone, than changes occurring only on a few files.

![Fig. 2. Complexity of a change period](image)

566
III. DEFECT PREDICTION FOR HARDWARE

Hardware repositories have similar features to software repositories, e.g., source code VCS, defect data base, specifications.

While adapting ECC, we assumed that hardware designs have generally a lower frequency of source files commits on VCS compared to software, so that the sampling period of 1 or 2 week as observed in [2] and [3] would probably be too short. As a starting point and reference we thought 2 weeks would be nevertheless interesting and that we could tune it later on. Based on the results of [2] and [3] we assumed that ECC with a decay factor for entropy would be the most promising as a low frequency of changes in a module is often a sign that is has reached a certain degree of maturity. It would also offer the possibility to tune the decay factor.

We measured the predictive power of our model like in [2], calculating the Spearman Correlation coefficient[5] between the ranking outputted by our model and the actual defect ranking based on our defect database over a set period of time. The length of this set period is also used as a tuning parameter.

We summed up our implementation in Fig. 3. We first preprocess the data using our VCS xml log export function (1). This log contains information such as the file name, the version of the file, who committed it and when. We then parse the xml log files using Ruby to retrieve only the information we are interested in (2)(3). From then we build two dimensional hash in which every period label contains each module’s name that has been modified and how many times it has been modified within the period(4). From this table we apply our ECC model and calculate the evolution of complexity in function of time(5). We finally compare the ranking of defect proneness outputted by our model at various points in time and compare it to the actual defect ranking coming from our defect database(6). Our code fits in less than 1000 lines of Ruby code.

We also applied variation to [3] described in [2] by replacing the number of code changes by the sum of line added, removed and changed during a time period as input metric.

IV. RESULTS

Our experiment methodology relies mainly on comparing elements extracted from past data, namely defect prediction results at a past certain point in time with the actual defect database status after this point. It is very similar to [2].

Our target for adapting defect prediction with ECC is about 77k lines of Verilog code. It is composed of 84 files, each file corresponding to a module, with an average of 924 lines of code(LOC) per file. The biggest file has 6.5k LOC. The source is maintained using a popular version control system. We believe one of the specificity of our project compared to software is that source code commits on the repository are done for the purpose of releasing it to the team of verification engineers which we talked about in the introduction. In other words, only module code having reached a minimum maturity is committed to the VCS. This minimum is often defined as the module being fully working in standalone but non-tested when integrated inside a “core”, which is the higher level of hierarchy, composed of more than ten modules. Therefore we believe the commit frequency in our project was much less frequent than on a typical software project. We have in average less than 10 commits per week spread on the modules which constitute our core. As of now, we do not differentiate changes due to defect fixes and those due to specification extensions.

Our defect database is maintained by proprietary technology, and includes typical information such as who found the defect, when, the kind of defect, when it was fixed and so on. For our experiments we focus exclusively on the date a defect was found and its location in the design. We consider the frequency of defect found in our case low compared to software. This is due to the fact that our design inherits significantly from our previous design and only defects found when integrating modules at the core level or above are reported in the system. We have an average of less than 5 defects found per week.

We used Spearman Correlation to compare the ranking of predicted defect prone modules by order of likeliness of still having defects and the actual defect order provided by the defect database to evaluate the predictive power of our model. We define a certain point in time as “present” and look at the tool output of our tool based on the “past” data relative to present. To establish a defect ranking from the actual defect data base, we define a “future” frame starting from “present”. We fine-tuned the “future” length to determine for which time period our tool is the most accurate. This is described in Fig4 where you can observe ECC evolving for 4 different modules inside our design. The grey area represents the time period for which we will establish the module defect ranking in our defect database.

We shifted the “present” point across the whole duration of the project. Each time the previous “future” point became the new “present” point and so on recording every time the Spearman Correlation coefficient. In the end, we calculated the average of all coefficients over this span and tuned our model to improve it.

We sum-up our results in table 1 for ECC and ECM Spearman correlation.
Compared to D’ambros [2], Spearman correlation is higher than the highest they obtained (on Equinox software) which is below 0.6. Of course, we need to test our tool on other hardware projects to be able to tell if we were lucky this particular project fitted well the model or if hardware in general is a good fit for the model.

One other interesting point is that, contrary to our hypothesis, we obtained the best Spearman Correlation for code change sampling with a period of 13 days, which is very close to software results described by D’Ambros.

Finally, we wanted to estimate how much verification time could ECC help us save by planning our verification by focusing on the modules with the highest entropy. Fig. 5 shows the evolution of normalized ECC as well as normalized cumulated defects over time. After peaking, ECC goes down monotonously with a light rise at last period. To estimate how well planning verification according to ECC could allow us to finish verification faster, we aimed to have the last defect free period to start earlier, as the length of this defect free period is one of the factor used to judge when verification is finished. To achieve this, we decided to push the period of times with no defects and low complexity at the end of schedule. In Fig. 5 we found 3 periods namely period 33, 34 and 36, with low complexity and 0 defects found that we pushed at the end of our verification plan. The result can be seen in Fig. 6 where these 3 periods are respectively period 35, 36, and 37.

Three periods compared to the time spent verification which spans over 25 periods of verification represent thus a possible 12% reduction of total verification time.

<table>
<thead>
<tr>
<th>TABLE I. SPEARMAN CORRELATION RESULTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spearman Correlation</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

We successfully adapted a software repository mining technique to hardware. We obtained promising results which showed that defects can be predicted and thus managed more efficiently in large scale hardware projects. Our future work will involve finding a method to schedule verification based on ECC so that verification ends sooner. Then we would try our defect prediction tool on several other hardware projects to measure its accuracy as well as measuring the significance as well as the explanatory power of our model. More importantly we will measure how much verification time it allows us to save.

REFERENCES


