Continuous Linting with Automatic Debug

Daniel Hansson
Verifyter AB
Lund, Sweden
daniel.hansson@verifyter.com

Abstract—Lint tools analyze RTL statically and report code segments that do not comply with the selected coding guidelines. It is quick to run and as the error messages are very precise it is easy to fix the issues. It requires much less resources to fix a linting issue than to find and fix the same issue during simulation of a test. However, the large amounts of error and warning messages that the linting tools produce is a problem.

Developers tend to push out linting until the very last moment just before the release. The engineers have plenty of critical test failures to attend to first, so it is a rational decision. However, this does not use the full potential of linting, because linting errors are very quick to fix, and some of the reported linting issues will be the same issues that takes much longer to debug during simulation. Pushing linting to the end of the project misses the opportunity to save a lot of time, but on the other hand developers cannot waste time by running the lint tool every night and again just-in-case they would find some good bugs.

The solution is what we call continuous linting, where we combine the linting tool with an automatic debug tool of regression failures. At the start of the project we select the subset of the coding guidelines that the design must comply to. The automatic debug tool then runs the linting tool regularly, e.g. each night, and when a linting issue is detected then the automatic debug tool sends a bug report to each individual with a list of linting errors this person has caused and in which commit they were introduced.

We believe this unleashes the full potential of linting. By reporting the issues immediately to the individual that caused them instead of waiting to the end of the project, issues are fixed earlier at no extra human effort. The issues have to be fixed at some point and better sooner than later. This is all made possible by the automatic debug tool. As the engineers will fix their linting issues immediately, this will probably save time as some of the simulation bugs will be fixed faster and with less effort as more issues will be fixed due to linting error messages and fewer by manually analyzing simulation failures.

In this paper we present the experience from a real ASIC project using continuous linting. We measured the number of linting issues that the automatic debug tool reported over a two month period. At the end of this period there were 0 linting issues outstanding. Had we waited until the end of the project to look at the lint issues there would have been 3617 lint issues in total to analyze.

This is bound to have a significant positive effect on the project. The cost of fixing linting issues immediately by the person that just caused them is much less than for an engineer to sift through large amounts of lint errors and warnings at the end of the project.

Keywords—continuous linting; automatic debug; regression testing

I. INTRODUCTION

During product development of ASIC’s or software, new bugs are continuously introduced by mistake causing the quality of the product to deteriorate. These bugs are called regression bugs. Today regression bugs are captured by running regular test runs, so called regression tests, which typically are RTL simulations that are run once or several times per day. The output of regression testing is a list of tests that fail. In a more advanced setup an automatic debug tool (e.g. PinDown) maps these test failures to the change that caused them and automatically sends a bug report to the engineer that made the bad change.

Regression testing does not have to consist of only RTL simulations. Other types of regression testing includes regular measurements of performance, synthesis area and power consumption. If the measurement is below certain threshold then this is defined as a regression test failure and is treated as any other regression test failure in the flow. Another less common regression setup is based on linting and that is what this paper is focus on.

Lint tools (such as Spyglass or Leda) analyze RTL statically and report code segments that do not comply with the selected coding guide lines. The error and warning messages that are produced during linting are very precise, they point to the actual line and describes what the problem is. Consequently it is very quick to fix linting issues. The problem is that these tools produce a large amount of errors and warnings, where some may reveal critical bugs, but the vast majority reveals minor issues. It is very labour intensive to go through all errors and warnings, which is why this is normally something that is done once at the end of the project. However, this misses the opportunity to fix major bugs at very little cost. We cannot let engineers manually sift through the linting issues on a daily basis as that would be too time consuming. The solution is continuous linting, where the power of linting is combined with automatic debug.

Continuous linting, which is a concept introduced in this paper, is a setup where an automatic debug tool (e.g. PinDown) runs linting at regular intervals, e.g every night, and maps the list of lint errors and warning to individual commits into the
revision control system. The automatic debug tool automatically sends the bug reports to the person who made that change. Engineers don’t need to spend any time on linting, unless they modified the code that caused a linting error or warning. These have to be fixed anyway, but as they are immediately alerted they can fix them right away instead of at the end of the project, costing nothing extra in terms of time. In fact the linting issues are easier to solve as they just made the modifications that caused them and don’t need any time remembering what they did long time ago, which is the case when you leave it until the end of the project. Fixing a linting issue that you just caused is a very precise task, compared to going through thousands of errors and warnings at the end of the project, most of which will probably have been caused by someone else. However the largest benefit is that a quick fix of a linting issue will sometimes fix a difficult simulation bug further down the line saving a lot of time.

This paper investigated a real commercial ASIC project that had a continuous linting setup, using the Synopsys Leda lint tool and the PinDown automatic debug tool for regression tests. At the start of the ASIC project a sub-set of all available coding guide lines where selected. Some effort was made during the start of the project to make sure that the linting rules were cleaned up, either by changing the RTL or removing coding guide lines that were ambiguous in terms of whether they actually required an RTL fix.

II. METHODOLOGY

A. How the measurement was done

We manually analyzed all regression lint bugs, i.e. commits that introduced new lint errors and warnings which were not there before. This was done by examining the bugs reports from PinDown (see Figure 1).

![Figure 1. Automatic Bug Report from PinDown, with information about the bad commit debugged by PinDown and the lint errors coming from Leda.](image1.png)

We examined a two month section of the ASIC project and noted down how many regression lint bugs where open and how many lint issues these encompassed. One commit can cause many lint issues, partly because more than one mistake may be introduced in the same commit and partly because the tools is very verbose and can report many lint errors and warnings which all have the same root cause.

We also noted all the times when PinDown reported 0 lint issues, which was most of the time.

B. What we did not measure

We did not measure how much faster lint issues where solved because they were automatically debugged by PinDown compared to if they had been manually debugged during the length of the project. The effect of automatic debug compared to manual debug has previously been analyzed in another paper\(^6\), but for functional RTL test failures. However for linting, manual debug during the project is not a viable option. Either the debug is automated which means problems are fixed during the project, or the debug is manual in which case it is pushed to the end.

We also only measured during a two month period of an ASIC project as we were only able to extract data for this time. The idea to write this paper was made after the project had finished and at that time not all data was available. Going forward it would be interesting to measure the impact continuous linting has on an entire project. The data set in this paper is too small to reach an overall conclusion, but it shows an interesting tendency, that would be interesting to study further.

There were regression bugs found during the project that came from RTL simulations. These have not been included in this paper as this was outside the scope for this lint-focused paper.

III. RESULTS

In total 5 regression lint bugs where found over the 2 month period we measured from 21st Feb to 20th April. Many other regression bugs were found but only 5 that caused lint errors or warnings. These regression lint bugs where all fixed as soon as they came in so we never saw an accumulation of regression lint bugs; we had 0 or 1 open regression lint bugs at any one time (see Figure 2).

![Figure 2. Open Regression Bugs with lint issues](image2.png)

Each regression lint bug caused 1 or more lint issues, the worst offender in total 1945 lint issues (a large commit). These lint issues were solved within a day most of the time, but the worst offender went from 1945 lint issues, first down to 30 lint issues before being resolved completely after a couple of days (see Figure 3).

The important result is that lint issues were fixed as they came in, normally within a day, and in the end there were 0 lint issues outstanding.
How does this compare to the standard methodology of doing the linting at the end just before release? In this case the linting issues would have been left until the end of the project before being addressed. Figure 4 shows the number of accumulated lint errors and warnings that would have been left until the end of the project: 3617 issues.

Another interesting aspect is that designers reacted to the bug reports as to any other reported bug. If an automatic debug tool reports to a user that the commit they just made caused an error, they just fix them. This is very different to how linting issues are normally fixed. They are normally pushed to the end of the project together with other chores such as code reviews, which in the end may not be done at all.

IV. Summary

Although the data set is too small to reach an overall conclusion, the tendency is clear. In this real commercial ASIC project we saw that linting issues were fixed much faster, most often within a day, with a continuous linting setup compared to the standard process of addressing linting issues at the end of the project, i.e. months later.

This dramatic improvement in fixing linting issues earlier was done without extra human effort as the debug was automated. The actual issues reported by the lint tool had to be fixed, it was just a matter of when in the project to fix it.

This is bound to have a positive effect on the overall project time schedule. Some of the linting issues are trivial, but some may be causing issues that are much more time consuming to debug in simulations or on prototype boards, thus saving a lot of time by catching them early.

REFERENCES