Finding Efficient On-Chip Test Pattern Sets

Adina Shanholtz, and Dr. Jennifer Dworak
Summer Research performed for the CRA-W DREU Program
At Southern Methodist University, Dallas, Texas USA

Abstract

Over the summer I worked on a project to generate and analyze on-chip test pattern sets that detect defects and wear out in hardware. These test patterns can be applied optionally in a multicore chip or in a 3D stack. To find the most efficient patterns, we investigated the generation of test patterns by identifying the mandatory detection conditions for hard faults and then filling the other inputs with pseudo-random values. The mandatory conditions are necessary, although not sufficient for the detection of the corresponding faults. By merging mandatory conditions, we can create patterns that test for multiple hard faults.

1. Introduction

Our research team focused on generating test sets that would be well-matched to the workload of a particular user in the field. Test patterns may be applied to chips in the system to detect circuits that have begun to fail due to wearout, something that happens over time. Our goal was to find an efficient way to detect not only all of a circuit's faults, but especially the most critical and hard to detect faults for a particular workload. This information can be used to optimize test sets so that these hard to detect and critical stuck-at faults can be detected when testing resources are very limited, as they would be for on-chip test in the field [Shi 2011].

Hardware monitors are used in many circuits to gather information regarding how many times an event happens in a circuit at runtime. However, they have historically not been used for providing gate level fault coverage information. I worked on developing methods that would utilize information provided by a hardware monitor that is inserted into a circuit design during manufacturing to analyze how many times a particular fault is likely to be detected. This type of hardware monitor captures partially-defined states of the machine and counts how many times each partial state was seen while executing specific instructions. The information provided allows us to focus on-chip testing resources on faults that are most important for the way a chip is actually being used [Shi 2011]. The focus of my project this summer involved developing new ways to automatically generate those test sets on-chip using the processing capabilities of an on-chip core.

2. Generating Test Sets to Be Applied On-Chip, Utilizing Information from Partial State Monitoring

2.1 The process

Our test patterns were generating by using templates that corresponded to a subset of the mandatory conditions required for the detection of several faults. Specifically, each template corresponded to a set of input assignments, where some inputs were assigned deterministic values while other inputs were left as "don't cares." The deterministic assignments were created by merging several hard faults' mandatory conditions. Hard faults were defined as those faults that were detected "n" times in an "n" detect test set (our data used n=15). To create the final pattern set, each template was replicated multiple times, and the "don't care" values were filled with pseudo-random values. The hope was that the resulting test set would do a good job of detecting all faults, especially those considered most critical for a given workload.

In order to test these pseudo-randomized strings, we needed to find a way to run them through Fastscan, the commercial software we were using for fault simulation. I wrote a script to automate the following process steps:

- filling the don't care values in each template with pseudo-random values,
- writing a test bench in Verilog to run through Modelsim, a Verilog simulator, in order to get the good circuit output values for each of the test patterns created from the templates,
- concatenating the input and output into readable files for Fastscan,
- running Fastscan and compiling the results into a fault dictionary.

This process took about four weeks to fully code and debug.

We ended up with a lot of data from about three different files so I wrote a program to compile it into a readable format. We now had information on how many times a specific fault was detected and how many faults a

pattern detected. With this information I could generate an updated probability table with more accurate results and find out how many patterns to generate in order to detect the hard fault with the smallest probability of random detection.

2.2 Problems with the Process

A good amount of the problems we ran into were with the commercial software we used.

After we got the automation working with real data, we had to double check to make sure we did not make any errors in the process. I found a discrepancy when checking to make sure the total number of faults was the same in our various documents. We spent an unfortunate amount of time retracing the automated process trying to find the problem, only to find that my original program that generated randomized test patterns was not randomizing the amount of times specified at input, but printing out only one randomized string that amount of times. Fixing that problem only gave us more accurate Fastscan results and the discrepancy remained.

Only after painstakingly going through the data did we find that Fastscan was not consistent with the data it gave us, and the program we had that compiled a fault dictionary did not handle Fastscan's inconsistencies. Once we fixed this problem with the fault dictionary we ran into a couple of problems with Modelsim. These problems were fixed with a simple solution, however they took a while to troubleshoot, as no one had a good background in the program.

When we finally got a good generation of data, we found another strange discrepancy in the file Fastscan gave us. According to Fastscan's output file, it detected certain faults multiple times with different patterns, which would mean that it was not "dropping faults" from the list of faults. If that were the case, it would mean that every pattern would (at minimum) have to detect at least one fault on each circuit output. However, the file Fastscan gave us had patterns that detected zero faults, which cannot happen. This last error was not solved while I was still working on the project.

3. Results

Once I was able to get data from my programs, I ran randomized test sets in groups of 10, 50, 100, 1000, and 5000. Only when we randomized the 8 partial states 5000 times did we get complete coverage of the circuit we were testing. Obviously these results are terrible, as one cannot load 40,000 individual patterns on to a chip for that one specific circuit. I was able to see from the results that after the initial detection of the easy faults, only one pattern out of every couple hundred would be effective, and end up detecting up to 50 hard faults. I concluded that we needed either more specialized pattern templates or perhaps to even hardcode certain patterns to detect the hardest faults.

4. Extra Work

In addition to the automated process for gathering data, I also spent some time rewriting the process for generating partial states. It was previously written in matlab, however not many people can read matlab so I rewrote the process for generating mandatory conditions and merging those conditions into partial states in C.

5. Conclusions

In this paper, we have taken the next step in gathering data using the method of partial state monitoring. Going even further, we could analyze the updated probability tables to find the hardest to detect group of faults, and choose which need to be hardcoded and which could be detected randomly. Eventually this process could be tested in the hardware itself and be able to focus on which faults appear after actual chip wear (as apposed to simulation).

6. References

[Shi 1] Y. Shi, K. Kaewtip, W.-C. Hu, and J. Dworak, “Partial State Monitoring for Fault Detection Estimation,” 2011 IEEE