VLSI System Testing Term Project Problem 1
Group 2

Kai-Hsiang Hu
B09901153

Chia-Cheng Wu
B09901098

Kai-Hsiang Chiu
R12943163

December 27, 2023

Abstract

In the early stages of semiconductor production, yield (Y) results– the ratio between working ICs in a wafer, are often low and unstable. This necessitates a high fault-coverage (T) to maintain an acceptable defect level (DL). As production processes stabilize and yields improve, there is an opportunity to reduce testing costs. Due to the rising of such demands, test set pruning has become a key strategy for yield improvement and reduction in total time required to test a batch of integrated circuits. By way of pruning the test pattern set, manufacturers can significantly cut costs associated with test time and equipment, without compromising the defect level (DL).

1 Introduction

1.1 Defect level curve

The relationship between fault coverage can be observed in the Figure 1. As yield increases, less fault coverage is required to achieve the same defect level, indicating that there are optimization opportunities.

1.2 Project Outline and Objective

In this project we were given 3 circuit designs with their respective fully specified pattern set. The circuits are named: s400, s9234, s38584 respectively, which indicates the circuit's size. The main objective is to reduce as much of the pattern set which is still able fulfill the arbitrary fault coverage demands of: 60

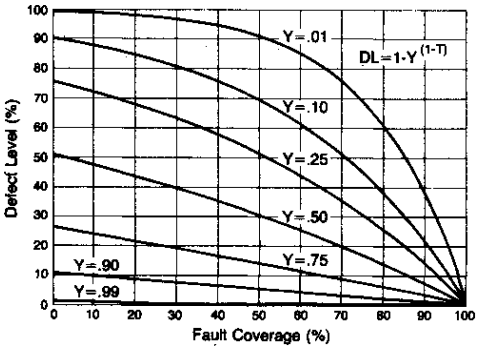

Figure 1: Defect level as a function of fault coverage.

2 Preliminary

2.1 Definition

In this paper, a set of Patterns is represented by an uppercase letter, e.g. P. The individual elements of this set are expressed as lowercase letters, for example, $p_i \in P$. An instance of the set P is denoted as $\vec{p}$. The size of set P is denoted as $|P|$. The function $g(P)$ produces the corresponding pattern detected faults, which are represented as $< g_1, g_2, ..., g_{|P|} >$, where $|g_j|$ denotes the number of faults detected by p_j. An instance of the function $g(P)$ is denoted as $\vec{g}(\vec{p})$.

Similarly, a set of Faults is also represented by an uppercase letter, e.g. F. The individual elements of this set are expressed as lowercase letters, for example, $f_k \in F$. An instance of the set F is denoted as $\vec{f}$. The size of set F is denoted as $|F|$. The function $h(F)$ produces the corresponding patterns that detect the faults, which are represented as $< h_1, h_2, ..., h_{|F|} >$, where $|h_l|$ denotes the number of patterns that detects f_l. An instance of the function $h(F)$ is denoted as $\vec{h}(\vec{f})$.

2.2 Single Pattern Fault Simulation

In the problem, we are given 3 circuits and their respective fully specified pattern lists. For each circuit with pattern set P, we apply *single pattern fault simulation(**SPFS**)* to the circuit fault list F, which can be done by applying fault sim to each pattern via TetraMax. After that we can get the faults detected by each pattern, and the patterns that detects each fault, they respectively represent the functions $g(P)$ and $h(F)$.

3 Proposed Techniques

3.1 Overview

The main technique that we adopted follows figure 2, our algorithm flow chart. We first use ***SPFS*** to identify which faults each pattern detects (blue box). Then we apply *Pattern-Fault Heuristics* and *Pattern Pruning Greedy Algorithm* to implement our main algorithm (green boxes). The time and memory usage are also recorded for our algorithm execution to be completed.

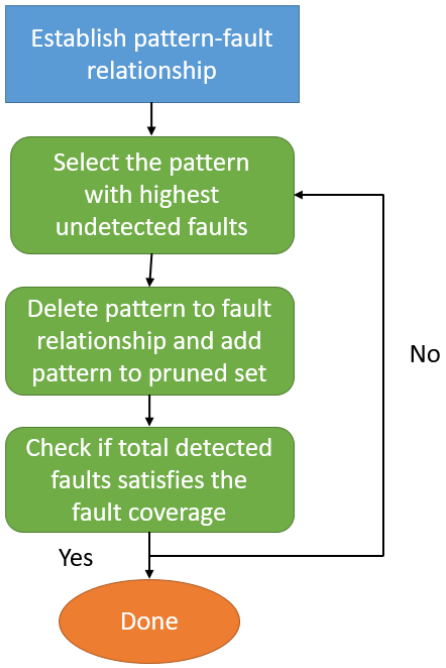

Figure 2: Flow Chart of Our Algorithm.

3.2 Pattern-Fault Heuristics

The first step in our algorithm involves the application of *Pattern-Fault Heuristics*. This heuristic is designed to identify patterns that contribute significantly to fault detection. The idea is to prioritize patterns that detect a large number of faults. By doing so, we aim to retain patterns that have a higher impact on fault coverage.

The ***SPFS*** provides information on the faults detected by each pattern. Leveraging this information, we compute the number of remaining faults that can be detected by each pattern and prioritize patterns with a higher fault-detection capability. This heuristic helps in selecting patterns that are more likely to contribute to achieving the desired fault coverage.

3.3 Pattern Pruning Greedy Algorithm

The core of our proposed technique is the *Pattern Pruning Greedy Algorithm*. This algorithm iteratively selects and prunes patterns based on their contribution to fault coverage. The algorithm follows the steps outlined in the flow chart (Figure 2).

1. **Initialization:**
   - Read the fault information from the file, including total faults, faults detected by implication, and origin pattern set P_{origin}.
   - Perform *Single Pattern Fault Simulation (SPFS)* to obtain the fault-pattern associations of faults detected by simulation.
2. **Pattern-Fault Heuristics:**
   - Identify pattern in P_{origin} that contribute most significantly to fault detection by prioritizing patterns with the highest number of detected remaining faults.
3. **Greedy Pattern Pruning:**
   - While the desired fault coverage is not reached:
     - For the remaining fault list $F_{remaining}$, select the pattern p_i with the maximum number of detected remaining faults and add it into our pruned pattern set P_{pruned}.
     - Remove p_i and $\vec{f} = \vec{g}(p_i)$ from P_{origin} and $F_{remaining}$
     - Remove the fault-pattern associations $\vec{h}(\vec{f})$ and update the detected remaining fault counts for other patterns accordingly.
     - Update the fault information and check if the desired fault coverage is achieved.

The proposed technique aims to efficiently prune the pattern set while maintaining the desired fault coverage. The greedy algorithm focuses on patterns that contribute the most to fault detection, ensuring an effective reduction in testing costs.

4 Experimental Results

4.1 Fault coverage with purned patterns

To effectively summarize the experimental data, we can categorize it by circuit, detailing the original number of patterns, the fault coverage percentages, and the pruned patterns for each specific circuit. Figure 3 adeptly presents this data for each circuit, illustrating both the fault coverage and the quantity of pruned patterns at various stages. Additionally, Figure 4, through its line graphs, demonstrates how fault coverage fluctuates in response to changes in the number of pruned patterns. Moreover, Figure 5, Figure 6, and Figure 7 showcase the execution of ATPG in Tetramax for each circuit, s400, s38584, and s9234, allowing us to juxtapose our experimental findings with those obtained from Tetramax across different levels of test coverage.

Circuit	Original Patterns	Fault Coverage (%)	Pruned Patterns
Test_s38584	675	61.422	3
		72.348	6
		80.7479	12
		90.075	36
Test_s400	39	65.7371	2
		72.7092	3
		81.2749	5
		91.0359	9
Test_s9234	166	66.9863	3
		71.4605	4
		80.8245	8
		90.1886	21

Figure 3: Table with original patterns, fault coverage percentage, and pruned patterns

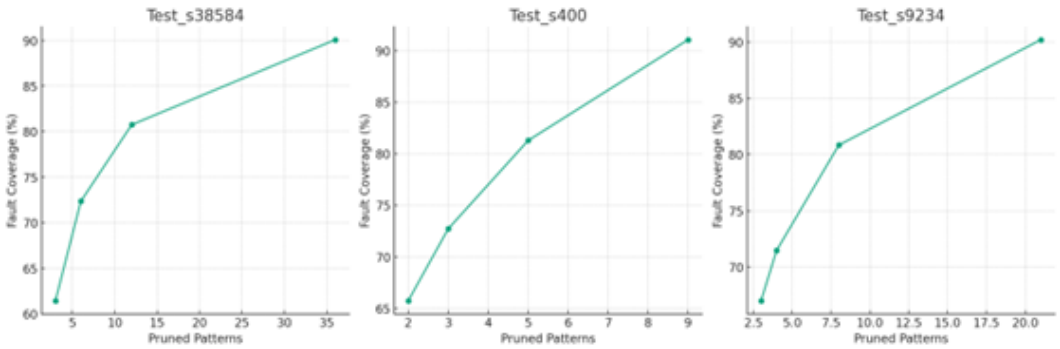

Figure 4: Line graphs for fault coverage and purned patterns

4.2 Fault coverage with CPU time

In Figure 8, we observe a relationship between fault coverage and CPU time. As fault coverage increases, CPU time similarly rises. However, this increase is not directly proportional to fault coverage. Notably, there are occasional reductions in CPU time at certain points within the fault coverage range. This observation leads us to speculate that the most time-intensive aspect involves identifying patterns that can cover the maximum number of faults. As fault coverage expands, the number of remaining faults diminishes rapidly, thus requiring less time for processing.

5 Conclusion

In our work we have discovered that used time is more related to the circuit size and less correlated to the designated fault coverage. Moreover, our proposed method had made a trade off between execution time and minimizing pruned pattern set.

We believe that such trade offs is worth while due to the fact that once the test pattern set is minimized, the more chips are tested with this predetermined fixed pattern set, the more time it would save in the future when compared with non minimized pattern sets. That is to say, our decision is more suitable for mass implementation with higher upfront cost which are smoothed out with the amount of chips manufactured and tested.

#patterns stored	#faults detect/active	#ATPG faults red/au/abort	test coverage	process CPU time
Begin deterministic ATPG: #collapsed_faults=340, abort_limit=10..				
27	189 151	0/0/0	75.10%	0.00
50	73 78	0/0/0	87.50%	0.00
78	40 38	0/0/0	93.90%	0.00

Figure 5: Results of s400 circuit in Tetramax

#patterns stored	#faults detect/active	#ATPG faults red/au/abort	test coverage	process CPU time
1696	139 15067	0/0/0	59.69%	0.82
1728	109 14958	0/0/0	60.03%	0.83
1760	86 14872	0/0/0	60.32%	0.84
2879	65 11788	0/0/0	69.86%	1.15
2911	88 11700	0/0/0	70.14%	1.16
2943	91 11609	0/0/0	70.45%	1.16
4349	65 8405	1/0/0	79.72%	1.50
4381	76 8329	1/0/0	79.95%	1.52
4413	72 8257	1/0/0	80.18%	1.53
4445	58 8199	1/0/0	80.34%	1.55
6747	45 4536	2/0/0	89.97%	2.12
6779	40 4496	2/0/0	90.08%	2.12
6811	45 4451	2/0/0	90.19%	2.13

Figure 6: Results of s38584 circuit in Tetramax

#patterns stored	#faults detect/active	#ATPG faults red/au/abort	test coverage	process CPU time
Begin deterministic ATPG: #collapsed_faults=2215, abort_limit				
30	1724 491	0/0/0	84.21%	0.00
60	231 260	0/0/0	91.64%	0.00
88	114 146	0/0/0	95.30%	0.01
117	74 72	0/0/0	97.68%	0.01
146	39 33	0/0/0	98.94%	0.01
166	29 0	4/0/0	100.00%	0.01

Figure 7: Results of s9234 circuit in Tetramax

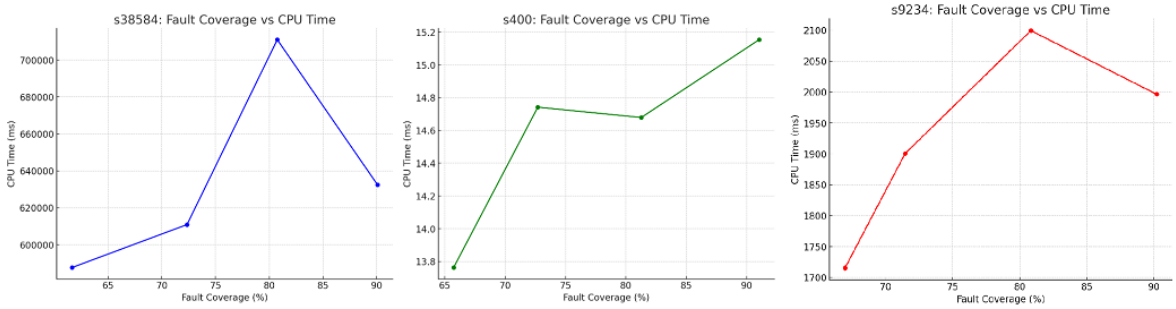

Figure 8: Line graphs for fault coverage and CPU time