The Dissertation Committee for Shahrzad Mirkhani
certifies that this is the approved version of the following dissertation:

Statistical Methods for Rapid System Evaluation under Transient and Permanent Faults

Committee:

Jacob A. Abraham, Supervisor

Andreas Gerstlauer

Lizy K. John

Subhasish Mitra

Michael Orshansky

Statistical Methods for Rapid System Evaluation under Transient and Permanent Faults

by

Shahrzad Mirkhani, B.S.; M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2014

Dedicated to my family...

Acknowledgments

Here, I would like to take this opportunity to thank the ones who have helped me during this step and every important step in my life. First and foremost I want to thank my advisor, Professor Jacob Abraham, for accepting me in his research group, encouraging me to do the research that I like, and giving me brilliant ideas for my research. I have learnt a lot from him, not only in VLSI area, but also in other aspects of life. I am also thankful to Professor Subhasish Mitra for having me involved in his exciting research projects. Working with him defined the true meaning of hard working to me.

I appreciate my committee members, Professor Lizy John, Professor Michael Orshanski, Professor Andreas Gerstlauer, and Professor Subhasish Mitra for their time and valuable comments on my dissertation. I would also like to thank Professor Nur Touba, who gave me the opportunity to start my Ph.D. program at the University of Texas at Austin.

I would also like to thank the ones that I have had a chance to collaborate during my Ph.D. pursuit, Bill Eklow from Cisco Systems, Hyungmin Cho from Stanford University, Dr. Chen-Yong Cher from IBM T. J. Watson research center, and Eric Cheng from Stanford University.

My research demanded working with several tools and creating an environment for millions of simulation passes. This was not possible without generous help and support from Andrew Kieschnick and Mary Matejka. I also thank Ms. Melanie Gulick, Ms. Debi Prather, and Ms. Melissa Campos. I acknowledge the Texas Advanced Computing Center (TACC) at The Univer-

sity of Texas at Austin for providing HPC resources that have contributed to the research results reported within this dissertation.

My gratitude extends to former and current members of Computer Engineering Research Center, Whitney Wadlow, Sriram Sambamurthy, Tung-yeh Wu, Hyunjin Kim, Jaeyong Chung, Ameya Chaudhari, Hsung-Cheng Lee, Ricardo Ramirez, Mahesh Prabhu, Eun Jung Jang, Kihyuk Han, Hyunsun Um, Jason Hu, and Balavinayagam Samynathan. I am specially very grateful to Balavinayagam Samynathan for all his efforts and ideas on our latest research.

I was very fortunate to have wonderful friends in Austin. They cheered me up when I was disappointed from my research and took care of my son during evening classes and meetings. First, I want to thank Sahar Ayazian-mavi and Behnam Robatmili who are more like family to me. I am also very thankful to my friends Newsha Mirzaei, Maryam Salimpour, Fatemeh Panahi, Ghazal Dashti, Leila Moravej, Georgina Vega Krum, Kavita Balakavi, Arda-van Pedram, Parisa Razaghi, Maryam Mortazavi, and Pantea Mirzaei.

Finally, and most importantly, I am deeply thankful to my family. I appreciate all that my mother, Mahin Safaei, has done, and still doing for me to follow my dreams. For this, she has given up her own dreams since I was very young. Without all the guidance, helps, and encouragements from my brother, Shahram Mirkhani, I would not have even chosen computer engineering as my career. And this dissertation is dedicated in memory of my father, Javad Mirkhani, who passed away four days after I came to Austin to start my Ph.D. program. I want to specially thank my son Shahriyar, who has shown a level of patience beyond his age in busy and tough days during this journey. And last, but not least, I am thankful to my lifelong friend and my husband, Maysam Lavasani for encouraging me to follow my dreams, helping me move

on from the most difficult days in my life, and never giving up on me. Not to mention that his technical ideas always expedited my research. Had we not met 19 years ago, I would have not been able to reach this milestone.

Shahrzad Mirkhani
December 2014, Austin, TX

Statistical Methods for Rapid System Evaluation under Transient and Permanent Faults

Shahrzad Mirkhani, Ph.D.
The University of Texas at Austin, 2014

Supervisor: Jacob A. Abraham

Traditional solutions for test and reliability do not scale well for modern designs with their size and complexity increasing with every technology generation. Therefore, in order to meet time-to-market requirements as well as acceptable product quality, it is imperative that new methodologies be developed for quickly evaluating a system in the presence of faults.

In this research, statistical methods have been employed and implemented to 1) estimate the stuck-at fault coverage of a test sequence and evaluate the given test vector set without the need for complete fault simulation, and 2) analyze design vulnerabilities in the presence of radiation-based (soft) errors. Experimental results show that these statistical techniques can evaluate a system under test orders of magnitude faster than state-of-the-art methods with a small margin of error.

In this dissertation, I have introduced novel methodologies that utilize the information from fault-free simulation and partial fault simulation to predict the fault coverage of a long sequence of test vectors for a design under test. These methodologies are practical for functional testing of complex designs under a long sequence of test vectors. Industry is currently seeking efficient solutions for this challenging problem.

The last part of this dissertation discusses a statistical methodology for a detailed vulnerability analysis of systems under soft errors. This methodology works orders of magnitude faster than traditional fault injection. In addition, it is shown that the vulnerability factors calculated by this method are closer to complete fault injection (which is the ideal way of soft error vulnerability analysis), compared to statistical fault injection. Performing such a fast soft error vulnerability analysis is very cruicial for companies that design and build safety-critical systems.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

As VLSI technology is heading to smaller feature size and more complex designs, the need for having more complex test plans and new techniques to reduce the cost of test becomes inevitable. These test plans include detection of both hard errors (e.g., manufacturing faults) and soft errors (e.g., transient faults). Time-to-market requirements are not satisfied by current methodologies for analyzing, with reasonable quality, a design in the presence of faults. Statistical methods can estimate the dependability of a design under hard and soft errors in a reasonable time with acceptable quality of results. Developing efficient statistical methods for this purpose is the main focus of this dissertation.

In 2001, the ITRS (International Technology Roadmap for Semiconductors) predicted that the cost of test will be more than the cost of manufacturing after 2012 [33], as shown in Fig. 1.1. During the past decade, there has been a great deal of effort to decrease the cost of test, and this is still an ongoing process. Techniques such as multi-site testing, adaptive testing, test compression, BIST (Built-in Self Test) and yield learning are the most important ways to decrease the cost of test, according to a survey done by the ITRS team in 2013 [35]. This report states, *"Significant progress continues in the reduction of manufacturing test cost, however much work remains ahead... Even though*

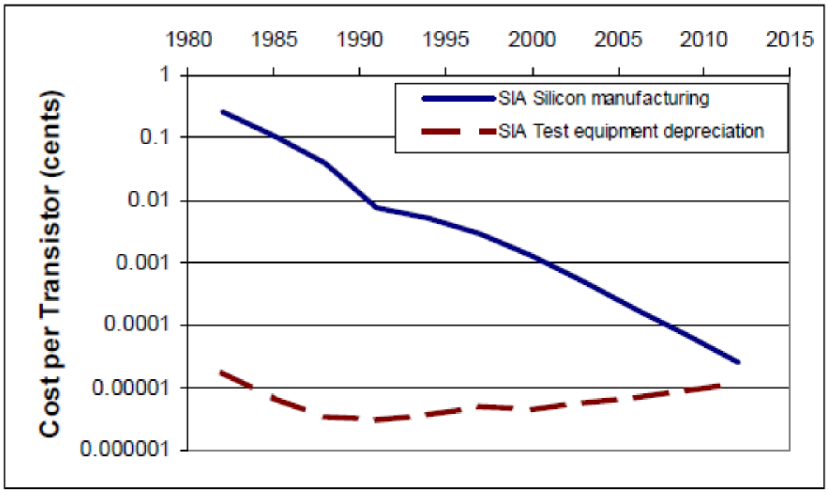


Figure 1.1: Microprocessor cost of test (from 2001 ITRS report [33], originally from ITRS 1997 report)

a lot of effort has gone into reducing the cost of test, 40% of the respondents to an extensive ITRS conducted survey from 2011 consider the cost of test as one of their biggest concern (compared to 30% in the 2009 ITRS roadmap survey) and 85% expect cost of test to become their biggest concern going forward...".

Automatic Test Equipment (ATE) and ATE interfaces, as well as test time and test coverage, are listed as the top cost drivers in current technologies [35]. As shown in Fig. 1.2, ATE cost is not scaled down in new technologies. In addition to ATE cost, data volume, necessary to test a chip with acceptable fault coverage is predicted to become one of the top cost drivers in the future [35]. Figure 1.3 shows the data volume is increasing exponentially each year [34]. Chips with typical complexity for the current technology use around 1 tera bits of (non-compressed) data for testing. Although test compression can decrease the amount of test data 2 to 4 orders of magnitude, the test time for either compressed or non-compressed test data is relatively high. The minimum required test data volume is shown in Fig. 1.4 [35].

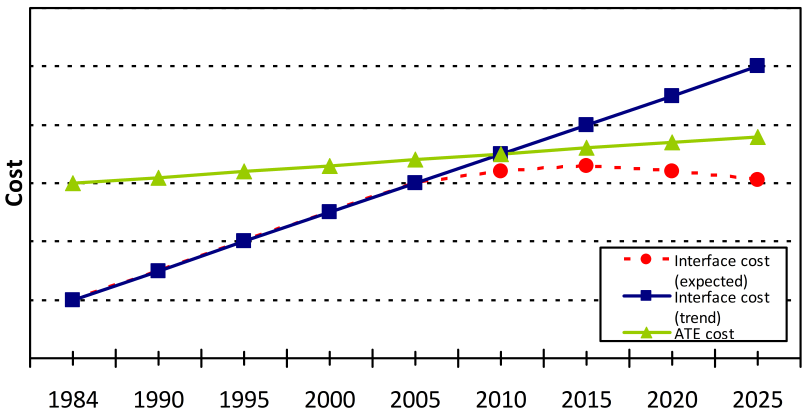


Figure 1.2: ATE cost (from 2013 ITRS report [35])

In order to overcome the above issues in testing, new methodologies should be developed to reduce the need for expensive ATE features. BIST techniques [20, 50] make chip testing possible without the need for expensive testers. In BIST, additional circuitry is added to generate pseudo-random [45] test vectors. These vectors are applied to the DUT (Design-Under-Test) and the responses are collected and compared to fault-free responses. For microprocessor testing, SBST (Software Based Self-Test) [18, 82, 83] has been proposed and originally called "Native mode self test". SBST tests a microprocessor by running software on the whole system. Using built-in self test methodologies in-field and at-speed testing will become possible [65]. At-speed testing has become more important in the past decade since it can capture more errors and malfunctions than the state-of-the-art scan testing [86].

Apart from the advantages listed above for methodologies such as BIST and SBST, there are some issues which make these methods hard to use. Generating efficient functional test vectors, which result in a high fault coverage for these methods is difficult and is usually done manually. On the other hand,

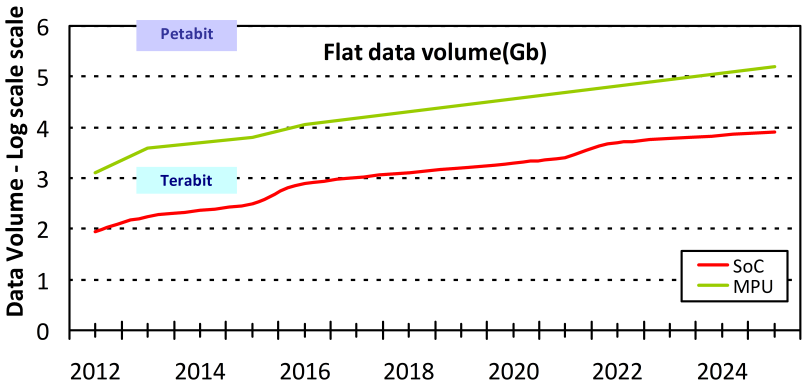


Figure 1.3: Test data volume (from 2012 ITRS report [34], quoted in [72])

if we use random functional test vectors, the initial coverage might not be adequate and there might be a need for additional iterations to improve these test vectors. To reach an acceptable fault coverage using functional test vectors, we might end up applying a large amount of test vectors to the design. According to Mentor Graphics [1], shown in Fig. 1.5, test data volume in at-speed testing is increasing with each new technology. If we are using simulation to evaluate functional test vectors, it will take an unacceptably long run time to evaluate these test vectors, since we need system-level simulation. If a prototype of the DUT can be programmed onto a FPGA (Field Programmable Gate Array) board, test vector analysis can be done orders of magnitude faster compared to simulation environment. However, programming a complex and large DUT onto a FPGA board might not always be possible.

One way to reduce the run time of evaluating functional test vectors (also known as functional fault grading) is to predict or estimate the fault coverage instead of performing detailed fault simulation. This method will

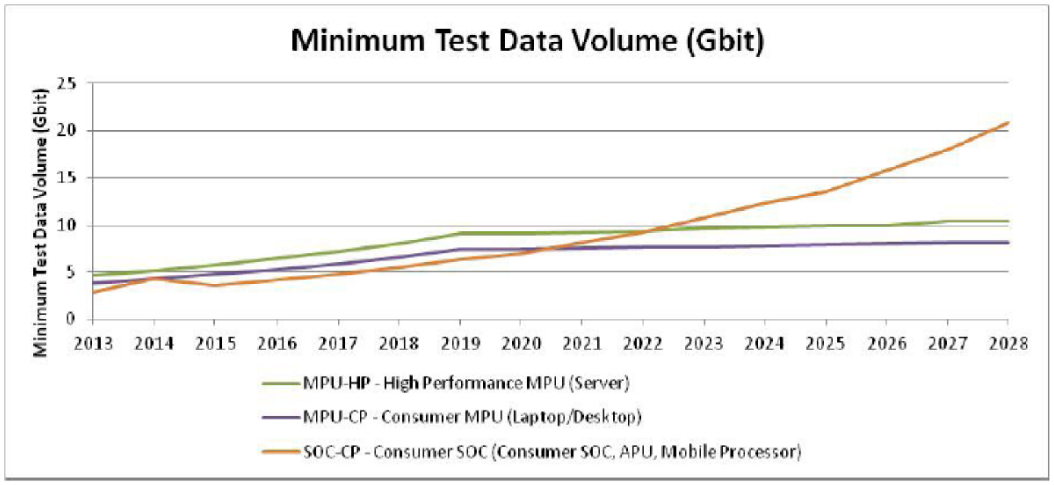


Figure 1.4: Minimum required test data volume (from 2013 ITRS report [35])

introduce inaccuracies due to the fault grading process. However, if the amount of error is not significant, coverage estimation can be very useful to rapidly estimate the fault coverage related to each sequence of functional test vectors and it can be used for further iterations of test vector improvement.

Apart from manufacturing faults, VLSI chips are also prone to soft errors [43, 57] mostly caused by cosmic rays. Soft errors do not have a permanent effect on the circuit and they are also called transient faults. However, they can affect and change the outputs of a program running on a system when they happen. According to ITRS 2013, which states *"Radiation-induced soft error rates are increasing to where, in addition to SRAMs, latches and flip-flops are likely to need protecting, at least for chips targeting enterprise applications. Other problems now increasingly observed include multiple-adjacent-cell SRAM upsets due to single radiation events and erratic shifts in minimum-operating-voltage, problematic for low power applications."*, the probability of soft errors occurring is increasing with shrinking transistor sizes. Analyzing

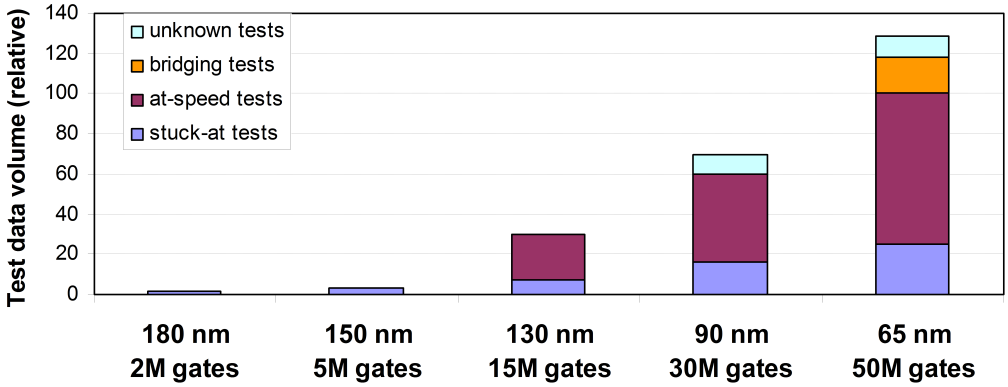


Figure 1.5: Required test data volume for at-speed testing (from Mentor Graphics [1])

the design for soft errors reveals the parts of a chip which are more vulnerable to cosmic rays, i.e., if a cosmic ray hits those parts of the chip it is more likely that the output of the system becomes different than the desired (also called golden) output. For life- and mission-critical systems, which should be highly dependable, preventing these errors from affecting the output of the system is very important. Some examples of these critical systems are avionics, medical devices, banking systems, and stock trading systems.

A similar problem to functional fault grading exists for analyzing soft errors in a large and complex design. The number of stimuli applied to a design to analyze the soft error vulnerability is very large, since typically these errors are analyzed while an application is running on that design. Another issue which makes soft error analysis even more involved than functional fault grading is the enormous number of soft error candidates that can potentially happen in a system. Since soft errors are transient faults, they can happen at any time during the application run. This demands that we consider millions or billions of these candidates in our analysis. Due to the fact that we need

to observe the output of a program under each error to be able to analyze the vulnerability of the system under soft errors, every error candidate must be analyzed in a separate application run, causing the need to run the application millions of times.

All the above-mentioned problems make a full vulnerability analysis for soft errors an impossible task to complete in hours or even days. Many of the current solutions for vulnerability analysis are based on statistical error injection [37, 73, 89, 92], which chooses a sample of error candidates for injecting into the system [40]. Based on the size of sample, we can define a margin of error (MOE) for the average soft error vulnerability of a design.

Similar to functional fault grading, the problem of soft error vulnerability analysis can also be accelerated by estimating the vulnerability, instead of extensive simulation used in state-of-the-art error injection methods.

With increasing the size and complexity of VLSI designs, current methodologies for analyzing a design in presence of faults do not satisfy time-to-market requirements. Statistical methods can estimate the dependability of a design under permanent and transient faults in a considerably short time. This dissertation focuses on three efficient methods to estimate fault coverage under a sequence of functional test vectors. Additionally, the problem of soft error vulnerability analysis is studied and a solution that uses statistical analysis is employed for a more accurate vulnerability analysis compared to traditional methodologies.

1.2 Background on Fault Grading

Fault grading has been studied for half a century [80]. Despite this strong background, the complexity of fault grading still makes it time-consuming for today's large designs. There are several reasons, including test quality, test cost, test vector re-usability, and test application time, that functional methods are becoming popular [48, 49, 59, 91], and these methods make fault grading problem even more complex. When design-for-test features, such as Logic BIST, are incorporated into the design, fault grading the resulting large number of test vectors becomes a difficult problem. Additionally, major companies are resorting to cache-resident software-based self test (SBST) schemes in order to apply effective tests for circuits with shrinking technology feature sizes and V_{dd}/frequency scaling [28, 65]. Furthermore, on-line test schemes for highly reliable systems require application-level test inputs while the system is operating in its functional mode.

Proposed methods attacking the fault grading problem, in general, can be categorized into three major groups: fault simulation, fault emulation, and coverage estimation (statistical methods). Approaches to fault simulation include basic gate-level algorithms [4] and several hybrid methods. These methods either combine basic methods (like PROOFS [63]), or use different levels of abstraction to speed up the simulation process [38, 51, 77]. Although these methods are faster than the traditional gate-level methods, they are still not scalable. To overcome this problem, high-level (e.g., Register Transfer Level or RTL) fault models [88] have been proposed. These methods scale well with the size of the design, but they lack a precise correlation with the fault models used to evaluate the coverage of manufacturing faults under test sequences. In general, in a large design with a large set of test vectors, system

level fault simulation seems to be nearly impossible [26].

Fault emulation was proposed in the 90s [94]. The main drawback of fault emulation is that the design under test should be fully synthesizable, which makes emulation not applicable in early design stages. Also, fitting large designs into the emulation hardware might not be feasible.

Another solution for fault grading, proposed first in the 80s, is to estimate the coverage using some data from good simulation and/or the circuit structure [5, 15, 39]. These methods are based on fault sampling [6], test vector sampling [29], and gate-level statistical analysis using data from design simulation (called STAFAN) [36, 44]. The initial work was mainly on the Single Stuck-At (SSA) fault model. Later, fault models were expanded to path delay faults and also sequential circuits [10, 30, 69]. Some high-level testability measurements were also introduced [75] and an extension of STAFAN to RTL components was proposed [76]. Since these statistical methods use good simulation data, their run time is comparable to the run time of good simulation, which makes them scalable. Apart from their scalability, such methods introduce some parameters (for re-convergent fanouts for example) which should be determined empirically. In addition, the results have been shown only for small circuits. There is also a commercial tool [9] based on statistical fault analysis which uses testability measurements and sets the faults with 0 probability as *undetectable*. Then it fault simulates the design with the rest of the faults. The error in this tool is claimed to be not more than 10%, but it still depends on the fault simulation on the whole system. Other methods for fault coverage estimation and test vector evaluation which use probabilistic distributions and fault modeling include the research in [21, 24, 36, 61, 81, 85] for fault simulation, and also for test vector evaluation [27, 46, 68, 93, 95]. Most

of these methods work for combinational/full-scan designs, or they need some parameters that might not be easy to calculate for every circuit (i.e., may not be useful for sequential circuits).

Therefore, in order to be able to test the designs using functional test vectors, there is a need for measuring the coverage for application level tests. For example, in microprocessors, these tests are instruction level tests (or some programs) which are applied to the the entire processor chip while it is operating in its normal mode inside the system. Industry designs have used such techniques as their test techniques [12, 65]. Chapters 2 and 3 of this dissertation address two different solutions to the problem of long run times when fault grading a design under functional patterns. However, although the methodologies in these chapters were aimed at working with functional patterns, these methods can be applied to both functional patterns and patterns for scan-based designs.

1.3 Background on Soft Error Vulnerability Analysis

Analyzing a design for radiation-based soft errors is an involved process, but necessary for life- and mission-critical systems. Results from [11] and [78] show that single bit-flips in the flip-flops of a system are suitable candidates for soft error modeling, since the system outcomes, when these errors are injected, are statistically close to the outcomes of a chip under actual radiation. Despite the fact that this model is accurate, analyzing a system for all soft error candidates (or even a sufficient sample of single bit-flips) takes a long time and a large amount of computational resources (e.g., processor cores). For a design with many thousands of flip-flops, analyzing each flip-flop's vulnerability, in order to add resiliency to the design, demands even more injections to reach

an acceptable margin of error.

During the past two decades, there have been methodologies developed to improve the run times for error injection of bit-flips in flip-flops. These methods are categorized below. A more detailed categorization and additional discussion can be found in [70].

- Hierarchical simulation methods that switch between different levels of abstraction, such as the transistor-level, gate-level, RT-level, and architectural level [13, 23, 74, 92].
- Analytical methods that use static analysis of circuits and probabilistic models of components (e.g., gates) to estimate soft error resilience of systems [8, 25, 32].
- Architecture-based methods that take advantage of architectural features and high-level (instruction-level) simulation passes to estimate the vulnerability [41, 58] or the worst-case soft error rate [60, 96] of the systems.
- System emulation that uses FPGA systems to build a prototype of the hardware that runs much faster than simulation [67].

Chapter 5 discusses a new methodology for vulnerability analysis of complex designs. The impact of soft errors has been increasing in new technologies, becoming a concern not only in mission- and life-critical systems, but also in general systems such as servers and routers. Therefore, analyzing a system in the presence of soft errors has (or should) become one of the stages in the chip manufacturing process. If, based on the costumer requirements, the designers find out that their design is vulnerable to soft errors, they need to

incorporate resilience techniques in their design to make it more dependable. In general, adding full redundancy is very expensive, and therefore designers usually apply partial redundancy to obtain an acceptable resilient system with a relatively low cost. In order to apply partial redundancy at the circuit level (e.g., using BISER [98]), designers need to know which parts of the system are more vulnerable to soft errors. Such an insight into soft error vulnerability requires a good model of soft errors. Several error models have been proposed for soft errors (e.g., [16, 17, 66, 71, 97, 99]). However, in this dissertation, this single bit-flip on flip-flop error model is used as the golden model[1].

Therefore, to apply partial redundancy to a design we need to have a means of determining which flip-flops in the system are the most vulnerable parts to soft errors. This calculation should ideally be done by injecting all possible error candidates and observing their outcomes under each error. Due to the enormous number of error candidates, which is the number of flip-flops in the design multiplied by the number of clock cycles a program takes to run on that design, this option is not feasible in a reasonable time. Another option for this calculation is *Statistical Fault (Error) Injection* or SFI [73]. In SFI, a sample of the error candidates is randomly chosen and the average outcome results for this sample can represent the average outcomes for all error candidates, with a margin of error (MOE) and a confidence level. Error injection with a sample of errors has been utilized to calculate the vulnerability of the flip-flops in a design in several papers [22, 67, 84]. In Chapter 5, the issues of detailed vulnerability analysis (per each memory element) will be discussed and it will be shown that the proposed methodology can result in more precise

[1]Soft errors are not considered in combinational logic in this dissertation, since soft errors in gates are decreasing in new technologies, such as multi-gate transistors [79].

detailed vulnerability than SFI.

The following chapters of this dissertation discuss my contributions to efficient functional fault grading and soft-error vulnerability analysis. As stated above, Chapter 2 describes a methodology for fault coverage estimation using local fault simulation. Chapter 3 introduces a new metric that can be used for estimating the fault coverage of a design, as well as an estimation method which uses this metric and regression method which is discussed in Chapter 4. Chapter 5 discusses another application of local fault simulation which can be used in estimating both the average and detailed vulnerability of a design under soft errors.

Chapter 2

Fault Coverage Estimation using Local Simulations

2.1 Overview

In this chapter, a method for manufacturing fault coverage estimation is discussed. This method is named FALCON (Fast fAuLt COverage estimatioN) [54]. FALCON has been applied on two processors, OR1200 [64] and IVM [90]. OR1200 has around 40,000 gates and 2,000 sequential elements, while IVM has around 5,000,000 gates and more than 100,000 sequential elements. Around 75,000 faults in OR1200 and 320,000 faults in IVM have been injected for this experiment. The experiments show that FALCON works much faster than fault simulation and it estimates the coverage more accurately than the fault sampling method [6] with a confidence of 0.998 [7]. The contributions in this work include the following.

- The idea of divide-and-conquer for coverage estimation of modular designs for coverage estimation has not been proposed before. Existing methods (e.g., STAFAN) use gate-level granularity rather than module granularity.
- A fully automated environment has been implemented for this coverage estimation method using a commercial fault simulator, a commercial logic simulator, and Perl scripts to feed proper inputs to these tools.

The structure for the remainder of this chapter is as follows. In Section 2.2, the approach is discussed. Section 2.2 describe the methodology. Then, in the following sub-sections (2.2.2, 2.2.3, and 2.2.4) we describe the details for each step of our estimation method. In Section 2.3, we show our experimental results. Finally, in Section 2.4, a brief run time analysis is discussed.

2.2 Coverage Estimation Methodology

This coverage estimation method has been designed to work for large modular designs, where a module can be combinational or sequential. A module boundary can be the HDL (Hardware Description Language) modules, such as Verilog, in a hierarchical design. However, if the design is flattened by the synthesis tool, an algorithm for partitioning the gate-level design is not difficult. Any boundary which includes a reasonable number of gates in a module can be used in this method. In Section 2.4, we provide an analysis which shows the relationship between FALCON run-time and module size. Since system-level feedback paths are one source of error in FALCON, it is better, in the partitioning process, to avoid having many system-level feedback paths.

In this chapter, the SSA (Single Stuck-At) fault model has been used. The main idea of this approach is to accurately estimate the coverage from the fault grading results on a standalone module, when that module is embedded in a larger system. This is accomplished by estimating how each *module* can propagate an error from one of its inputs to one of its outputs. Also, the probability of the presence of fault effects on the outputs of each module-under-test (MUT) is calculated. The latter factor gives an idea of how many fault effects will be activated on the boundaries of a MUT, while the former factor

helps finding how many of these activated faults can be propagated through other modules in the whole system. Combining these two, it is possible to estimate how many fault effects can reach the system's primary outputs.

The following are a few terms used frequently in this chapter.

- MUT (module-under-test): A module in the system which is the target for fault grading. FALCON performs fault grading module by module.
- Detection probability table: A table indicating the probability of each fault in a MUT to be present at one of the outputs of that MUT. There is one detection probability table for each MUT.
- Propagation table: A table indicating the probability of error propagation from one input of a module to one of its outputs. There is one propagation table for each module in the design.
- Local test vector set: The set of stimuli at the inputs of a module, when applying the test vector set to the primary inputs of the system.
- Stand-alone fault simulation: The process of fault simulating a module, separated from the system, with its corresponding local test vector set.
- Local fault dictionary: The resulting fault dictionary when performing stand-alone fault simulation on a module.

2.2.1 Algorithm Steps

This section describes FALCON methodology, step by step, using an example. The details of each step will be discussed in the following sections.

- Step 1: Given a test vector set, FALCON simulates the entire system to generate local test vectors for each module. This step is done using a commercial logic simulator.

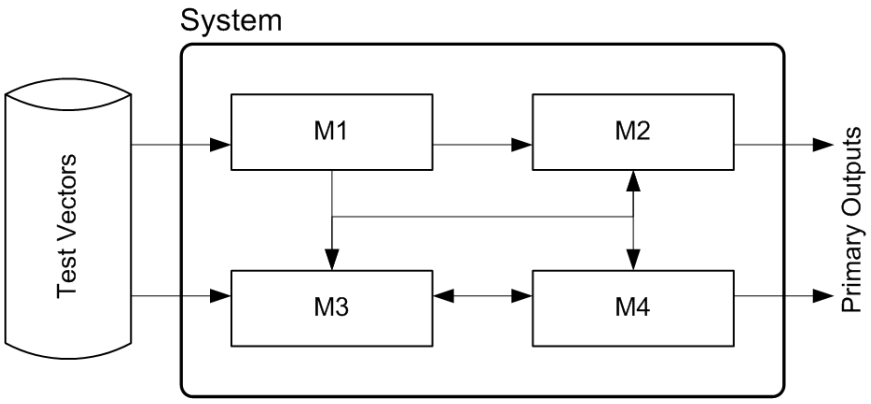


Figure 2.1: System block-diagram

A system with four modules and a set of test vectors is shown in Fig. 2.1, while Fig. 2.2 shows the system after applying this step.

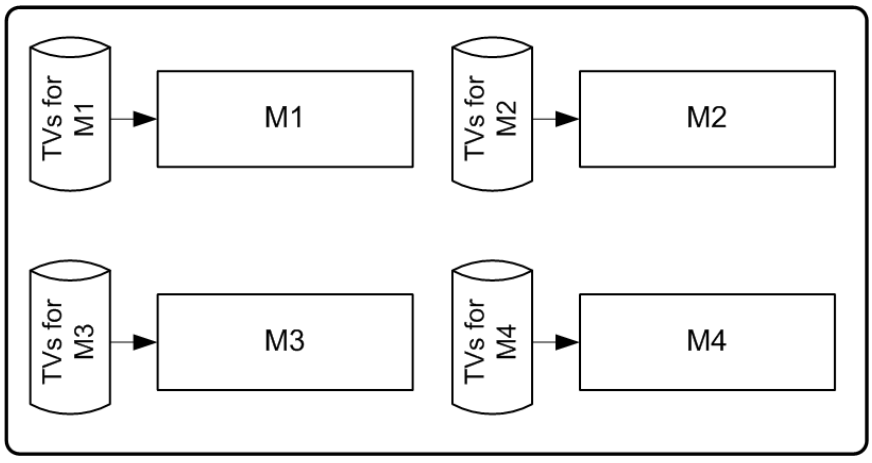
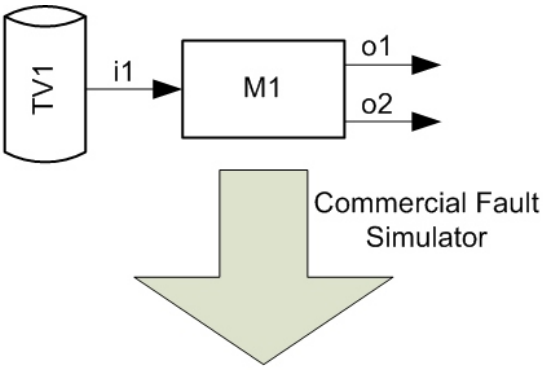


Figure 2.2: Module with local test vectors

- Step 2: Now that local test vectors are available, FALCON performs

stand-alone fault simulation for each MUT ($M1$ in the example). Note that faults are not dropped during this process, because the probability of each fault detection is calculated. Therefore, the more times a fault is detected on a MUT output, the higher the probability it can be detected on a system primary output. In this step, the results are stored in local fault dictionaries. This step is done by a commercial fault simulator (shown in Fig. 2.3).



	Fault	o_1 (0/1)	o_1 (1/0)	o_2 (0/1)	o_2 (1/0)
	f_1	10	2	0	0
i1 s@0	f_2	25	0	1	0
i1 s@1	f_3	13	0	2	0
	...				
	f_k	1	10	4	0
	...				

Figure 2.3: Local fault dictionary

- Step 3: Using the local fault dictionaries from step 2, detection probability tables are generated for each MUT and propagation tables are generated for all modules in the system (Fig. 2.4). Note that for modules which are not in the MUT set, propagation tables are still needed.

These will be discussed in more detail in Section 2.2.3. This step is done with a Perl script.

Propagation Table for M1

Input	Output	0/1 -> 0/1	0/1 -> 1/0	1/0 -> 0/1	1/0 -> 1/0
i1	o1	0.5	0	1	0
i1	o2	0.076	0	0.038	0

Detection Probability Table for M1

Fault	o_1 (0/1)	o_1 (1/0)	o_2 (0/1)	o_2 (1/0)
f_1	0.2	0.04	0.0	0.0
f_2	0.5	0.0	0.02	0.0
f_3	0.26	0.0	0.04	0.0
...				
f_k	0.02	0.2	0.08	0.0
...				

Figure 2.4: Propagation and detection probability tables

- Step 4: FALCON generates a statistical model using module interconnections in the design, propagation tables for each module in the design, and detection probability tables for each MUT (Fig. 2.5). By simulating this statistical model with a commercial simulator, it is possible to estimate the fault coverage of each MUT in the entire system. We will describe the probability calculation formula in Section 2.2.4.

2.2.2 Detection Probability Tables

As discussed above, this table indicates the detection probability for each fault on each output of a MUT. It is implemented as a 3-dimensional array; the first dimension represents the fault number, the second dimension

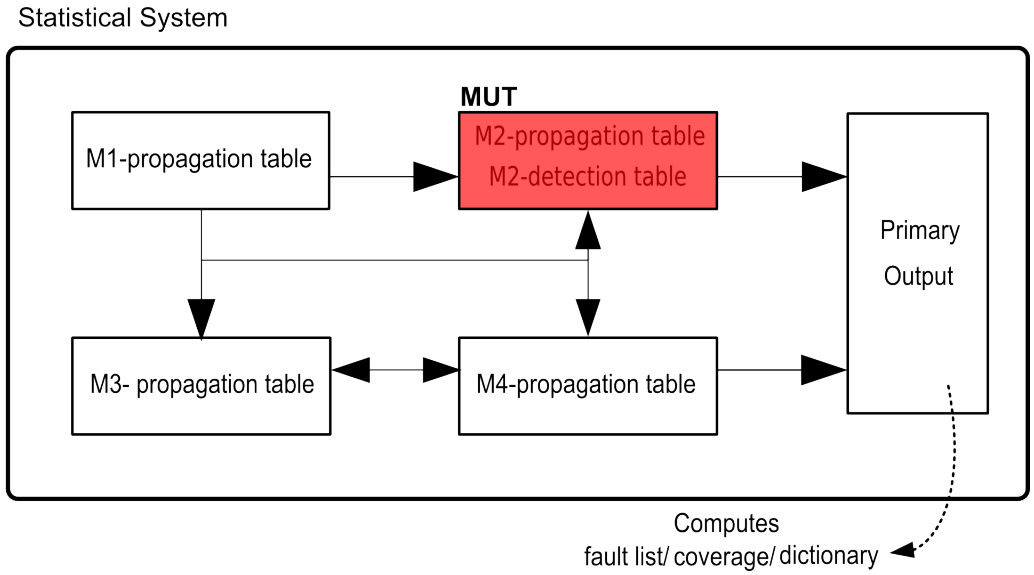


Figure 2.5: Statistical system block diagram

represents the MUT output number, and the third dimension is either 0 or 1. Zero represents a 0/1 value and one represents a 1/0 value (a line with $v/\bar{v}$ value shows that a fault effect has reached that line and inverted the value of that line from v to $\bar{v}$).

The value of each element in this table, e.g., $det\_prob\_table[f][o][v]$, is calculated as described below.

$$det\_prob\_table[f][o][v] = \frac{\#\ of\ times\ f\ is\ detected\ on\ o\ with\ value\ v/\bar{v}}{\#\ of\ test\ vectors} \tag{2.1}$$

As an example, fault #10 is detected on the 5^{th} output of a MUT in the stand-alone fault simulation process, 4 times with value 0/1 and 11 times with value 1/0. Suppose the applied test vector set contains 100 test vectors,

then the detection probability table includes

$$det\_prob\_table[10][5][0] = 0.04 \text{ and } det\_prob\_table[10][5][1] = 0.11$$

2.2.3 Propagation Tables

This table calculates the ability of a module to propagate a fault effect from each of its inputs to each of its outputs. As discussed before, this table is generated using the local fault dictionaries of each module. However, if the description of this module at the gate level is not available, this table can be generated by simulating this module stand-alone (with its local test vectors) and inject the module's input stuck-at-0 (stuck-at-1) faults by putting a constant 0 (1) instead of the value of that input. The number of simulations will be $2 \times i$ where i is the number of module inputs. Since this is done on a high-level module, the simulation cost is not too high. In another case, if the detailed gate-level description of a module is available, but there is no need to perform fault grading for this module, FALCON can inject only the faults on this module's primary inputs and perform stand-alone fault simulation.

Similar to detection probability tables, propagation tables are also implemented as a 3-dimensional array. The first dimension is the input number, the second dimension is the output number, and the third is a number between 0 and 3. Value 0 for this dimension shows the propagation probability of a 0/1 value from an input to a 0/1 value to an output. Table 2.1 shows the interpretation of other values for this dimension.

The value of each element of this array is calculated to be the *propagation factor* from an input to an output. If value $v/\bar{v}$ can be propagated through output o, it means that fault $i - sa - \bar{v}$ is detected on output o. Therefore, propagation factors are calculated from fault simulation as shown below.

Table 2.1: Interpretation of index values in propagation table

Index Value	representation
0	$0/1 \rightarrow 0/1$
1	$0/1 \rightarrow 1/0$
2	$1/0 \rightarrow 0/1$
3	$1/0 \rightarrow 1/0$

$$prop\_table[i][o][k] = \frac{\#\ of\ times\ i{-}sa{-}\bar{v}\ detected\ on\ o\ with\ w/\bar{w}\ effect}{\#\ of\ times\ i{-}sa{-}\bar{v}\ activated}$$

Note that the numerator is not divided by the number of test vectors. This is because whenever this factor is used in probability calculation, the fault effect has been already propagated through the input of this module. Therefore, we only needed to use a definition similar to conditional probability (i.e., the probability of a fault effect propagation given that the fault is activated).

Also, it should be noted that the number of faults detected can be more than the number of fault activations and this factor becomes greater than 1. This can happen in modules with sequential feedback paths. Therefore, "propagation factor" is a better term than propagation probability in this case.

2.2.4 Detection Probability Function

Suppose FALCON has generated a propagation table for each module in the system and it has generated the detection probability table for the MUT. Now, using these tables, we need to calculate the detection probability of each fault on the primary outputs of the system. For this purpose, a function needs to be defined that accepts the detection probability values at the inputs of a module and calculates the detection probability values at the output of that module, using its propagation factors.

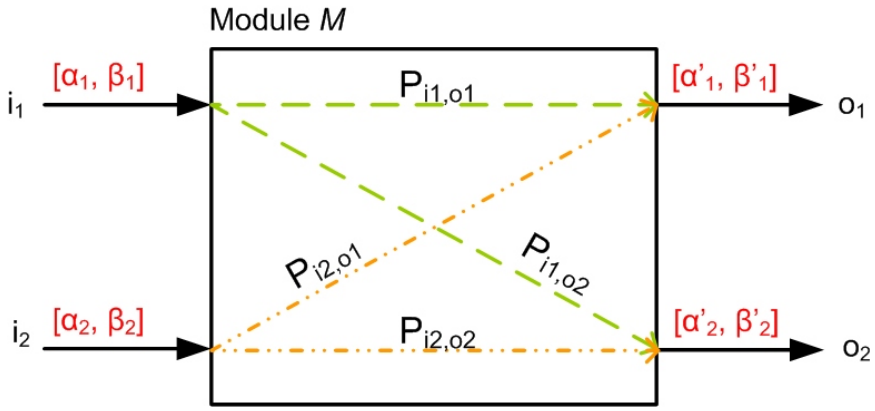


Figure 2.6: A sample for detection probability function

Suppose a fault effect is propagated through more than one input of a module. This case happens when there are system-level fanouts in the design. In our example in Fig. 2.6, suppose a fault effect has reached input i_1 and i_2 with 0/1 probability values equal to α_1 and α_2, and 1/0 probability values equal to β_1 and β_2, respectively. In this case, it is easier to calculate the probability of *absorption* of a fault effect from ALL inputs through an output and then negate this absorption probability to reach the propagation probability from either of inputs to that output. This idea is a realization of the following probability formula (suppose A and B are independent events),

$$P(A \cup B) = 1 - (1 - P(A)) \times (1 - P(B)) \tag{2.2}$$

Note that FALCON is adding some error by assuming that the two events are independent, because in reality, two inputs of a module can affect each other during fault effect propagation (i.e., the fault effect can be masked). Since this error happens only in case of system re-convergent fanouts and intra-module re-convergent fanouts are taken care of by stand-alone fault

simulations, only a small amount of error due to this assumption is expected in this estimation method. This is validated by the experimental results discussed in Section 2.3. Another source of error in FALCON is when the design has inter-module (i.e., system-level) feedback paths. The experimental results show a small amount of error in this case as well as masking errors.

Using Eq. 2.2, the detection probability of the fault effect on i_1 and i_2 reaching o_1 with value 0/1 can be calculated as,

$$\begin{aligned} det\_prob[o_1][0] &= \alpha'_1 \\ &= 1 - [(1 - \alpha_1 \times P_{i_1,o_1}^{0/1\rightarrow 0/1}) \times (1 - \beta_1 \times P_{i_1,o_1}^{1/0\rightarrow 0/1}) \\ &\quad \times (1 - \alpha_2 \times P_{i_2,o_1}^{0/1\rightarrow 0/1}) \times (1 - \beta_1 \times P_{i_2,o_1}^{1/0\rightarrow 0/1})] \end{aligned}$$

The other values for detection probability of the outputs can be calculated in a similar way. A general formula for o_1 with value 0/1, when a fault effect reaches N inputs is,

$$det\_prob[o_1][0] = 1 - \prod_{n=1}^{N} (1 - \alpha_n \times P_{i_n,o_1}^{0/1\rightarrow 0/1}) \times (1 - \beta_n \times P_{i_n,o_1}^{1/0\rightarrow 0/1})$$

2.2.5 Fault Detection Metric

Now that the detection probabilities of a fault on each line in the system can be calculated, there should be a way to determine which value (or

ranges of values) should be determined as **detected** and which ones should be considered as **not detected**. In other words, a metric for fault coverage is needed in FALCON.

Using FALCON statistical system and the statistical simulation environment (Section 2.2.6), it calculates the detection probability for MUT faults from the outputs of each MUT through the primary outputs of the system. Since detection probability is defined as in Eq. 2.1, detection threshold can be defined as,

$$detection\_threshold = \frac{1}{\#\ of\ test\ vectors}$$

This threshold means that the fault is detected one time when applying our test vector set to our design. Therefore, if a detection probability value at a system primary output is greater than or equal to this value, it will be counted as a detected fault.

Using the detection probability function and the detection threshold defined above, FALCON can estimate the fault coverage of the system for each MUT. Due to the detection probability definition in Equation 2.1, the output of the statistical system shows fault detection in the system **as if they are not dropped**.

2.2.6 Statistical System and Simulation

After building the propagation tables and detection probability tables, it is time to calculate the detection probability for each line in the design using the detection probability function discussed in Section 2.2.4. Note that if the top module of the system (the module that the statistical system is built from) has some glue logic, it is wrapped inside a dummy module and propagation

tables for this dummy module are generated as well. This process has been done in one of the discussed test cases in this chapter. FALCON statistical system (in Verilog) follows the steps below.

- Replaces every module in the system with its propagation table.
- Adds a detection probability table to the MUT.
- Connects these high-level models as they were connected in the original design.
- Changes the signal type to a type which accepts the detection probability for both 0/1 and 1/0 values (e.g., a two element array of type *real*).

Given the above statistical system (along with a library containing detection probability functions), and a commercial simulator, the detection probabilities of interconnections and system primary outputs can be calculated. For coverage calculation, detection probabilities on primary outputs are compared with the above-mentioned defined detection threshold.

2.3 Experimental Results

There are scripts developed for generating local test vectors and test-benches for stand-alone fault simulation to be able to apply FALCON on designs. Figure 2.7 shows the flow of our estimation methodology. As can be seen in this figure, the local test vector sets (TV_1, ...) are obtained using a commercial simulator. Then using a commercial fault simulator, local fault dictionaries are obtained based on the local test vectors (*Local Dict.* 1, ...).

These local fault dictionaries are converted to detection probability and propagation probability tables. These tables, along with the interconnections of the system are used in a simulation environment to estimate the fault detection probabilities in the whole system.

FALCON has been applied on two CPU designs, OR1200 which is an open RISC processor [64] and IVM which is an implementation of the DEC Alpha processor developed in University of Illinois at Urbana Champaign [90]. These CPUs are Verilog designs which were synthesized with the TSMC 180nm technology library. Table 2.2 shows some characteristics for each test case. These experiments have been run on an Intel® Xeon® X5670, 2.93GHz processor, with 72GB of memory, and 12 cores (with hyper threading). In both cases, fault grading is started after the design has been reset (using the reset signal of the design).

As discussed in previous sections, FALCON estimates the presence of each fault on each output of a design. This can be considered as a statistical fault dictionary. To show the accuracy of FALCON estimation method, sequential fault simulation on a sub-set of faults for OR1200 without fault dropping has been performed and the appearance of each fault on each primary output has been calculated. On the other hand, FALCON has been performed on the same sub-set of faults and the detection probability of each fault on each primary output has been calculated. An example is shown in Fig. 2.8 (fault simulation) and Fig. 2.9 (FALCON) for the faults in the ALU module in OR1200. As can be seen, these two measurements are very close to each other, which means that FALCON is able to prepare statistical data about fault detection rather than outputting only a coverage number. This data can be used for purposes like fault diagnosis. Other estimation methods,

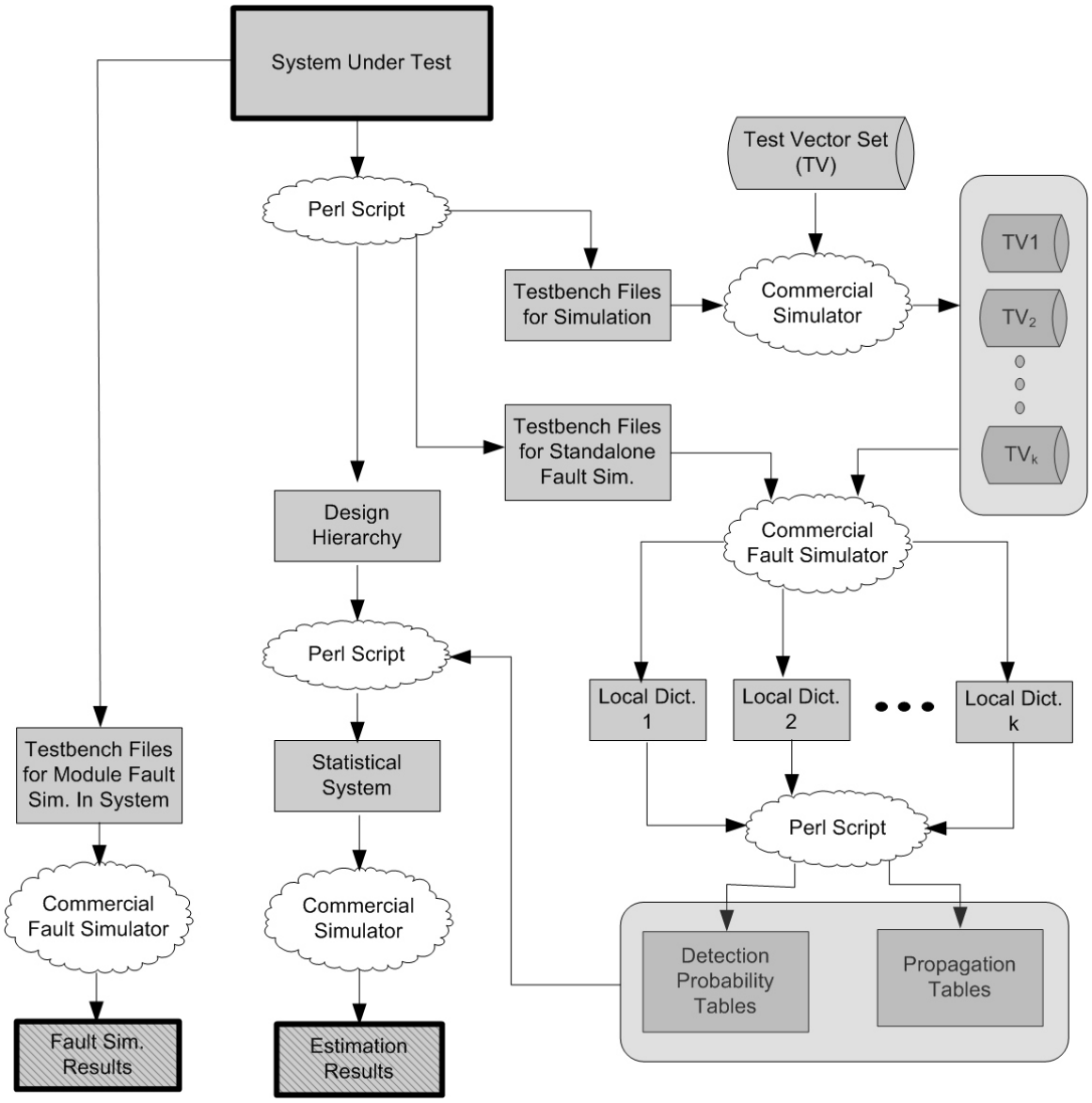


Figure 2.7: Coverage estimation measurement flow

like fault sampling, do not output any data other than the fault coverage. However, in this chapter, I have compared FALCON results with the results of fault simulation and fault sampling **with** fault dropping.

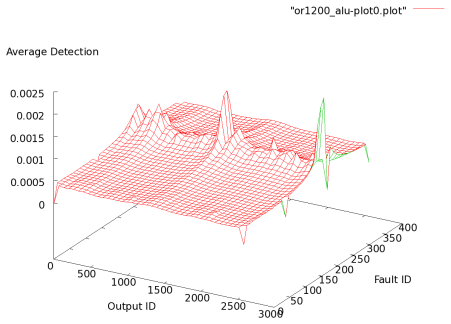


Figure 2.8: Fault simulated average detection

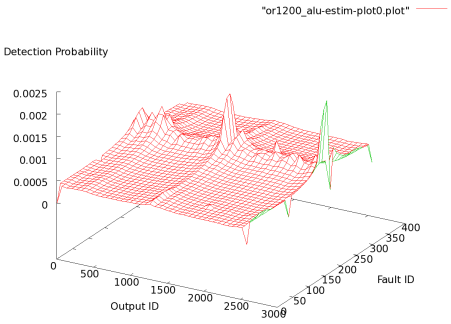


Figure 2.9: FALCON detection probability

Traditional fault simulation and fault sampling (using the Cadence Verifault-XL® fault simulator that was also used in FALCON for standalone fault simulations) has been done on the entire system, and the run times and coverages has been measured, and I have compared the run time and fault coverage of FALCON with these results. As discussed above, the fault simulation and fault sampling processes are done with fault dropping.

In th fault sampling method, a sample of faults from the fault list is selected and fault simulated. Based on the fault coverage from these sampled faults, the fault coverage for the whole system is calculated using a formula ($C_{3\sigma} = c \pm \frac{4.5}{N}\sqrt{1 + 0.44Nc(1-c)}$, where N is the total number of faults and c is the fault coverage calculated for a sample of faults). This method gives the user a range of fault coverage with a level of confidence. Based on a few experiments on both designs, the sample size which gave the most accurate estimations was as 10% of the whole faults and the confidence equal to 0.998

Table 2.2: Testcase characteristics

design name	approx. size (gates)	MUT modules	# of inputs	memory elements	analyzed faults
OR1200	40,000	13	387	2,000	75,129
IVM	5,000,000	18	836	100,000	320,912

(known as 3σ). For example, for OR1200 design, with 1024 test vectors and a sample of 7512 faults (10%), fault sampling method gives a range equal to [28.4, 30.9] with a confidence of 0.998. This means that with a probability of 0.998, the real fault coverage (for the whole system) is between 28.4% and 30.9%.

The experimental results show that the fault sampling coverage range does not match the real coverage in several cases. In cases that the calculated coverage matches the real fault coverage, 0% error is considered in the tables and figures. For the cases that the real coverage is not in the calculated range, the error is calculated as the difference between the real fault coverage and the coverage in the middle of the range.

In the following sections, two case studies are discussed.

2.3.1 OR1200 Case Study

For the OR1200 case study, random test vector sets with different sizes have been applied to the CPU. Fault coverages are shown in Tab. 2.3. The first column shows the number of test vectors (which are random), while columns 2, 3, and 4 show the coverage results for fault simulation, fault sampling, and FALCON, respectively. As discussed above, in column 3, a range of fault coverage is shown. Column 5 indicates the error between FALCON estimation and fault simulation method. The error is calculated as **the number of**

mis-calculated faults over the total number of faults. That is why the difference between fault coverages shows a smaller number than the error shown in the fifth row of Tab. 2.3. The sixth row of this table shows the error between the fault sampling method and the traditional fault simulation method (columns 2 and 3). As discussed above, the error is defined as 0% if the real coverage is in the range of the calculated coverage. The next two columns in this table show the number of mis-detected faults (rather than the percentage) in FALCON and fault sampling methods, respectively. The last column shows the difference between the number of mis-detected faults between fault sampling method and FALCON. As can be seen, this number is relatively high in the first three cases.

Table 2.4 shows the run time results for fault simulation, fault sampling, and FALCON for the runs whose coverages shown in Tab. 2.3. The first column of this table shows the number of random test vectors. The second, third, and forth columns show the run times of fault grading for fault simulation, fault sampling, and FALCON, respectively. In column 5, the speed-up in the run time between FALCON and fault simulation is shown. This speed-up factor is calculated by dividing the time spent in fault simulation by the time spent in all the steps of FALCON (i.e., column 2 divided by column 4). All run times are shown in *seconds*. We measured the run time speedup between FALCON and fault simulation. As can be seen in this table, fault sampling works faster than FALCON, but the error in the sampling method is more than that of FALCON in most cases, as shown in Tab. 2.3. Also, when the design size grows, the fault sampling method calculates less accurate results compared to FALCON. However, the sampling method run time is still comparable to FALCON. This can be seen in the IVM test case in the next section (Tables 2.5

Table 2.3: Fault coverage results for OR1200

# of test vectors	Fault sim. cov.(%)	Fault sampl. cov.(%)	FALCON cov.(%)	FALCON err.(%)	Sampl. err.(%)	FALCON mis-det.	Sampl. mis-det.	Mis-det. diff.
1,024	39.57	[28.4, 30.9]	38.99	0.96	9.92	722	7453	6731
2,048	42.99	[35.4, 38.1]	42.51	0.97	6.24	729	4689	3960
4,096	45.82	[38.1, 40.8]	46.65	1.46	6.37	1097	4786	3689
8,192	53.69	[52.4, 55.1]	54.39	2.34	0	1759	0	-1759
12,288	57.63	[59.0, 61.7]	58.42	2.24	2.72	1683	2044	361
16,384	60.1	[59.0, 61.7]	61.41	2.11	0	1586	0	-1586

Table 2.4: Run-time results for OR1200

# of test vectors	Fault sim. run time (s)	Fault sampl. run time (s)	FALCON run time (s)	Speedup (Fault sim. time/FALCON time)
1,024	846	100	386	2.19
2,048	1,608	130	679	2.37
4,096	4,341	387	1,488	2.92
8,192	3,788	751	1,470	2.58
12,288	15,608	1275	4,672	3.34
16,384	18,578	1792	6,125	3.03

and 2.6).

The OR1200 results have been summarized in Fig. 2.10. This figure shows the run time for fault simulation, fault sampling and FALCON on a logarithmic scale (shown with bars). Also, sampling error and coverage estimation error are shown in this figure with lines. These errors are shown as the number of miscalculated faults. As can be seen in this figure, the fault sampling method has the fastest run time when the number of test vectors is increased. It can be seen that the run time for FALCON grows more slowly than for traditional fault simulation. For the IVM test case, FALCON works faster than the fault sampling method. This is while only a small subset of faults have been used for fault grading. I believe that FALCON will run more efficiently than fault sampling with smaller error rates if more faults are injected in a large design.

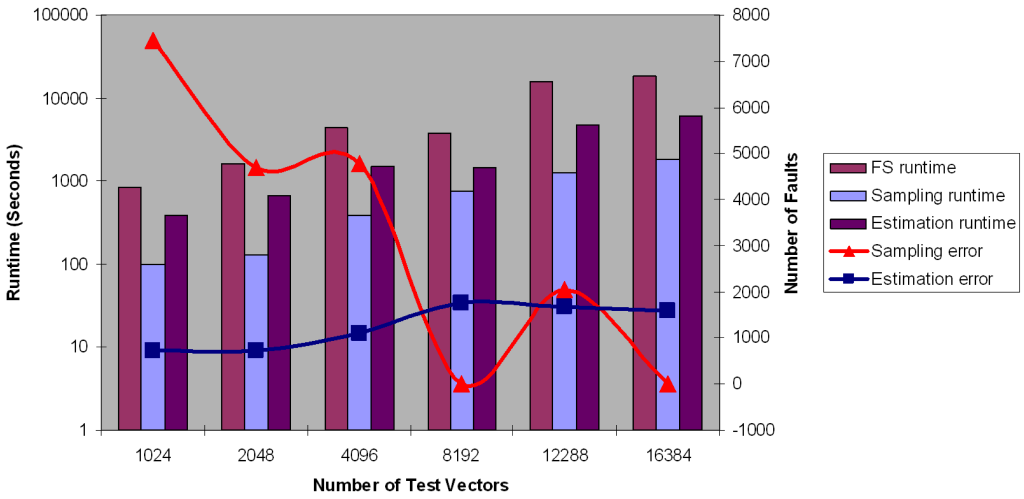


Figure 2.10: OR1200 run time results and mis-detected faults

2.3.2 IVM Case Study

In the IVM test case, random test vectors which are valid instructions are applied to the processor. Fault simulation, fault sampling, and FALCON results on IVM for test sequences with 50, 200, 500, 1000, 2000, and 5000 clock cycles can be seen in Tab. 2.5. Since IVM is a superscalar processor, the number of random instructions in the memory model is more than the number of clock cycles for which the design is fault simulated.

In this case, a subset of faults, and not all fault candidates, for this processor is chosen for injection. This is because the fault simulation process could not be finished in a reasonable time even for a small number of cycles when all the faults are injected in the circuit. Similar to the OR1200 case (Section 2.3.1), the results for coverage and run time for fault simulation, fault sampling, and FALCON are shown in this section. Fault coverage results

Table 2.5: Fault coverage results for IVM

# of test vectors	Fault sim. cov.(%)	Fault sampl. cov.(%)	FALCON cov.(%)	FALCON err.(%)	Sampl. err.(%)	FALCON mis-det.	Sampl. mis-det.	Mis-det. diff.
50	15.9	[19.6, 20.9]	15.3	0.82	4.35	2,632	13,960	11,328
200	21.8	[26.6, 29.1]	19.93	2.01	6.05	6,451	19,416	12,965
500	41.4	[47.4, 49.0]	39.46	2.1	6.8	6,740	21,823	15,083
1,000	45.0	[50.6, 52.2]	43.44	1.89	6.4	6,066	20,539	14,473
2,000	49.5	[55.51, 57.08]	48.4	1.21	6.8	3,884	21,823	17,939
5,000	53.4	[57.91, 59.48]	51.88	1.81	5.3	5,809	17,009	11,200

Table 2.6: Run-time results for IVM

# of test vectors	Fault sim. run time (s)	Fault sampl. run time (s)	FALCON run time (s)	Speedup (Fault sim. time/FALCON time)
50	13,841	3,273	610	22.6
200	30,057	6,370	1,232	24.3
500	77,833	11,665	4,762	16.34
1,000	111,984	13,762	8,910	12.5
2,000	243,780	52,851	18,795	12.97
5,000	477,859	61,188	40,090	11.91

are shown in Tab. 2.5 and run time results can be found in Tab. 2.6.

As can be seen in Tab. 2.6, in this test case, fault sampling takes longer than FALCON and as shown in Tab. 2.5, the error between fault sampling and fault simulation is higher than the error between FALCON and fault simulation (Tab. 2.5).

Similar to the OR1200 case, run times, fault coverage, and coverage errors for fault simulation, fault sampling, and FALCON are shown for the IVM processor. Figure 2.11 shows the run times for the three methods and errors in coverage for fault sampling and coverage estimation. The run times are shown in logarithmic scale and the error is indicated by the number of faults.

As can be seen in Fig. 2.11 for both the OR1200 and IVM cases, the run time in FALCON, due to its scalability, grows at a slower rate than fault simulation. Also, it can be seen that FALCON runs faster than fault sampling

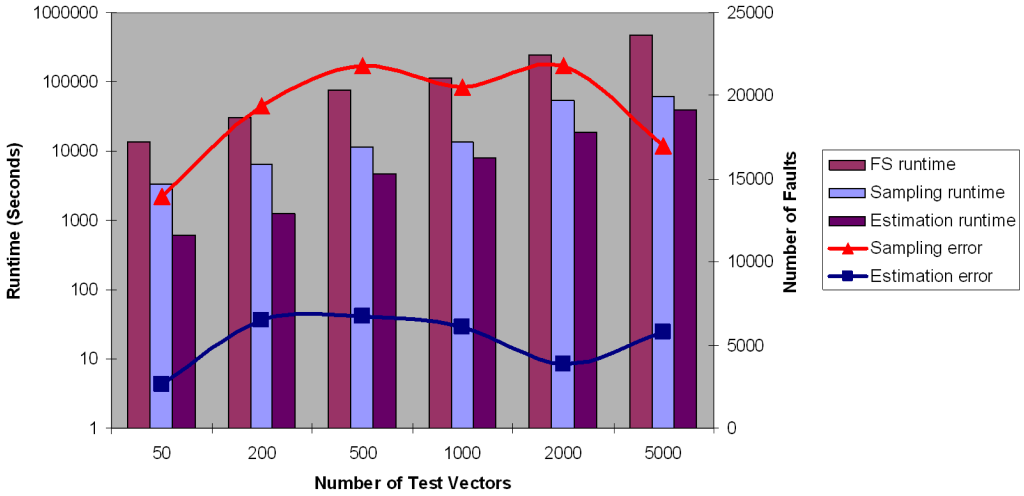


Figure 2.11: IVM run time results and mis-detected faults

with the growth of the design size, with smaller error rates.

One advantage of FALCON over sampling is that it can determine which faults are detected on which outputs. This is useful when the user needs more data than a simple coverage number (e.g., in the case of fault diagnosis).

As shown in the above two test cases, the run time of FALCON grows faster than fault sampling; however it is still faster than fault sampling for large designs. The main reason for this growth rate in coverage estimation is that the faults are not dropped during the local fault simulation process. This can give the user extra information about each fault detection, such as the probability of detection for each fault on each output. This can be considered as a statistical fault dictionary. In cases that the user does not need this extra information FALCON can divide the test vector set into sub-sets, and apply

the same process to each sub-set. In each step, it can drop the detected faults. This way, the time of the stand-alone fault simulations will be reduced. Also, smaller partitions can be used in FALCON so that the local fault simulations, without fault dropping, will be more efficient and take less time.

The above experimental results show that fault sampling is a great method for estimating fault coverage for small to medium designs. It is still a good way to roughly estimate fault coverage for larger designs. However, this method does not provide data other than fault coverage. On the other hand, FALCON works a lot faster than fault simulation. Although it works slower than fault sampling for small to medium designs, it becomes faster than fault sampling for larger designs. In addition, FALCON provides more information about fault detection which can be useful during the test process.

2.4 Run-time Analysis

In this section, a simple run time complexity analysis is shown for FALCON and it is compared with fault simulation run time.

The following parameters are used in this analysis.

- M: number of modules in the system
- T: number of test vectors
- f_m: number of faults in an MUT m
- G: number of gates in the system
- g_m: number of gates in MUT m
- $i_{max} \times o_{max}$: maximum module input/output product

Using the above definitions, the complexity of each step of FALCON can be expressed as follows, assuming that there are m modules in the system under test.

- Good simulation: $O(G \times T)$
- Stand-alone fault simulation: $O(g_m \times f_m \times T)$
- Generating the Propagation table: $O(I \times T)$,where I is total number of module inputs
- Generating the Detection probability table: $O(f_m \times T)$
- Forming the testbenches: $O(M)$ (this usually takes only a few seconds)
- Statistical simulation: $O(f_m\prime \times M \times i_{max} \times o_{max})$, where $f_m\prime$ is the number of detected faults in stand-alone fault simulation. In worst case $f_m = f_m\prime$

All of the above should be done for coverage estimation of module m. Therefore, the run time of FALCON can be written as,

$$O(G \times T + g_m \times f_m \times T + I \times T + f_m \times T + M + f_m \times M \times (i_{max} \times o_{max})) \quad (2.3)$$

If all of the faults are injected in a MUT, the number of faults is linearly related to the number of gates. Therefore, f can be replaced by g in formula 2.3. Also, $I \times T + M$ part can be removed since it is negligible compared to the other parts (more like a constant). The statistical simulation part $(g_m \times M \times (i_m \times o_m))$ is not negligible if all faults propagate through all

inputs of every module. Since each fault usually affects a limited part of the design, it will propagate through a few of the module paths. Therefore, using $i_{max} \times o_{max}$ in this formula is unrealistic since this term can be easily replaced by a small constant. On the other hand, M is also a relatively small number and the whole product of $M \times i_{max} \times o_{max}$ can be replaced by a constant. As a result, the estimation of run time can be written as,

$$Estimation\ run\ time = O(G \times T + g_m^2 \times T + g_m) \tag{2.4}$$

which can be written as,

$$Estimation\ run\ time = O(G \times T + g_m^2 \times T) \tag{2.5}$$

Equation 2.5 shows that the run time of FALCON mostly depends on the time for good simulation and the time for local fault simulation for module m.

On the other hand, the complexity of fault simulation for a module with g_m gates can be written as:

$$FS\ run\ time = O(f_m \times G \times T) = O(g_m \times G \times T) \tag{2.6}$$

If each part of Eq. 2.5 is compared with Eq. 2.6, it can be seen that fault simulation run time is proportional to $g_m \times G$, while coverage estimation run time is partly proportional to G, and partly to g_m^2.

An Example: Suppose a system under test has 5 million gates (G) and 10,000 test vectors (T) are provided for fault grading this system. If FALCON runs on a module with 50,000 gates (g_m), then the following run

time estimations for FALCON and fault simulation (based on Eq. 2.5) can be calculated. Since in complexity analysis we deal with the biggest exponent,

$$coverage\ estimation\ run\ time = c_{fal} \times max\{5 \times 10^{10}, 25 \times 10^{12}\}$$

or,

$$coverage\ estimation\ run\ time = c_{fal} \times 25 \times 10^{12}$$

On the other hand, based on Eq. 2.6,

$$fault\ simulation\ run\ time = c_{fs} \times 5 \times 10^{6} \times 5 \times 10^{4} \times 10^{4}$$

which can be written as,

$$fault\ simulation\ run\ time = c_{fs} \times 25 \times 10^{14}$$

where c_{fal} and c_{fs} can be considered as constants.

As can be seen, FALCON can work around **100** times faster than fault simulation. For example, if FALCON takes a few minutes (hours), it can be expected that fault simulation takes for *hours (days)*.

Due to the above analysis, if g_m is close to G (which means g_m is a large module in the design), FALCON will be as time-consuming as fault simulation. If there are such modules in the design, they need to be broken into smaller modules. Fortunately, with today's hierarchical designs, every module has its own sub-modules. Therefore, these sub-modules of large modules under test can be used and FALCON can be applied hierarchically to the design.

2.4.1 Memory Complexity

A commercial test tool typically uses a more efficient method than traditional serial or parallel fault simulation. In this case, it can be assumed that there is a fault queue for every gate during fault grading process. For a simple

memory consumption analysis, an average fault queue for each gate during the whole fault simulation process can be assumed as the main source of memory consumption. Therefore, the average memory used in a fault simulation process can be expressed as,

FS_mem = $(f_{avg} * G)$

where, f_{avg} is the average length of fault queue and G is the total number of gates in a design.

On the other hand, the memory consumed for FALCON is mostly due to stand-alone fault simulation and statistical simulation steps. Since each step is done separately, the peak memory consumption can be calculated as the maximum memory consumption for each step.

FALCON_mem = max(standlone_mem, statistical_mem)

stand-alone_mem = $O(f\prime_{avg} * g_{max})$

where, $f\prime_{avg}$ is the average length of fault queues in local simulation. In general, the average length of fault queues during a fault simulation pass depends on several factors, like the logic level (and the sequential level in case of full sequential fault simulation) of the module or circuit under test, the quality of the applied test vector, and fault dropping option. In this case, the logic and sequential depth of the module under test in FALCON is less than the logic and sequential depth in whole design for fault simulation. On the other hand, FALCON performs fault simulation without fault dropping, while in fault simulation faults are usually dropped when they are detected during the previous cycles. Therefore, depending on the size of the module, the quality of the applied test vectors (how many faults they can activate and how many of the fault effects they can propagate), and the reduction in logic

level, $f\prime_{avg}$ might be smaller or greater than f_{avg}.

and,

statistical_mem = $O(\Sigma(i_m * o_m) + f\prime * (i * o)_{max})$

where, the first term is for propagation tables and the second term is the MUT detection probability calculation in worst case. $f\prime$ is the number of detected faults in stand-alone fault simulation.

In the worst case,

statistical_mem = $O((M + f\prime) * (io)_{max})$

Since $f\prime$ is usually by far more than M, the above equation can be re-written as,

statistical_mem = $O(f\prime * (io)_{max})$

An Example: Like the previous example, suppose the design under test has 5 million gates (G), with an MUT of size 50,000 gates (g), which is 1% of G, and maximum $i * o$ equal to 500*500.

If f_{avg} is 20000 and the detected number of faults in stand-alone fault simulation is 30000, and $f\prime_{avg}$ is 25000, then,

FS_mem = $O(2 * 10^4 * 5 * 10^6) = O(10^{11})$

stand-alone_mem = $O(3 * 10^4 * 5 * 10^4) = (15 * 10^8)$

statistical_mem = $O(3 * 10^4 * 25 * 10^4) = O(75 * 10^8)$

FALCON_mem = $max(15 * 10^8, 75 * 10^8) = O(75 * 10^8)$

In this example, FALCON can potentially use 10 times less memory than fault simulation. In reality, memory consumption efficiency is better since memory consumption for FALCON was calculated in a pessimistic way

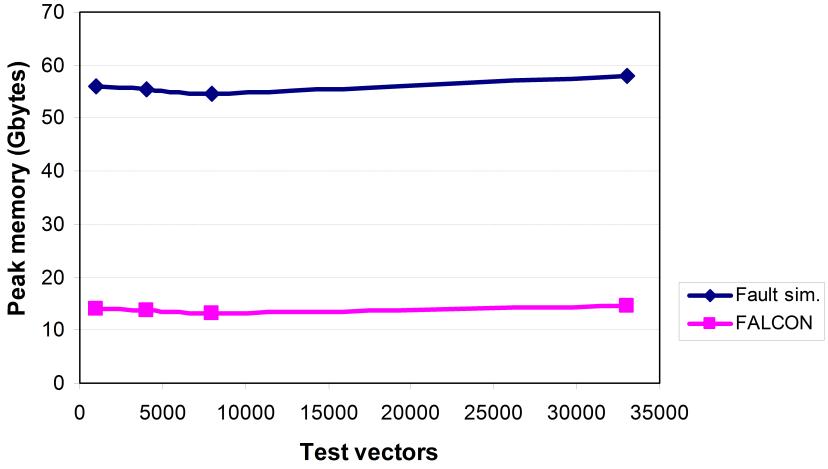


Figure 2.12: Peak memory of FALCON and fault simulation for IVM in case of inefficient sequence of test vectors

above. In addition, the memory consumption efficiency is higher during initial phases of test vector development, in which the coverage is not high. Figure 2.12 shows a case where inefficient test vectors were applied to the IVM design and it caused fault coverage less than 1%. Only in one case was the fault simulation completed and the rest of the IVM fault simulation cases did not run to completion and were terminated after two weeks. As can be seen in this figure, the memory consumption in FALCON is around 4 times less than the memory consumption in the (incomplete) fault simulation, while the biggest module used in FALCON, which is the memory unit, has around 250K gates (5% of G), more than 850 inputs, and more than 870 outputs. Therefore, it can be seen that the above analysis is pessimistic for FALCON.

In general, if logic and sequential depths in FALCON and fault simulation are known, a more accurate memory consumption analysis can be done.

2.5 Conclusions and Future Directions

In this chapter, a hierarchical and modular technique for estimating fault coverage (FALCON) was discussed. This method is evaluated for single-stuck-at faults. The experimental results show that for large designs, the approach proposed here can achieve orders of magnitude improvements in time with a very small amount of error. This method works the best when each module in the design is a few times smaller than the whole design. Large modules can be simply broken into smaller modules and FALCON can be applied on them hierarchically. FALCON works on both combinational and sequential modules. This method can be used even at early stages of the design, when there are only a few modules available at the gate level of abstraction.

FALCON is a general method that is not restricted to stuck-at fault mode. As a future work this methodology can be extended to other fault models, such as the delay fault model.

Chapter 3

GIC: A Metric for Fast Test Vector Set Evaluation

3.1 Overview

In this chapter, a new metric based on local signal value combinations is introduced [53]. It is shown that there is a high correlation between (single stuck-at) fault coverage and this metric for combinational and scan-based, as well as sequential circuits. Rapid test vector set evaluation using this metric needs one pass of fault-free simulation, and fault coverage estimation requires a partial fault simulation in addition to one pass of fault-free simulation, requiring much less time than full fault simulation. The introduced metric is very easy to measure and its generality makes it useful for any design. Although this metric works well with small circuits, it is especially useful for large circuits and/or very large sets of test vectors where the simulation time could be very large. The circuits under test can be gate-level or even standard cell based. However, in this chapter, our circuits are all gate-level.

The rest of this chapter is organized as follows. Section 3.2 introduces this metric, a method for estimating fault coverage and test vector evaluation, and some points that should be considered while using this metric. Section 3.3 explains the experimental results gained by measuring this metric on several benchmarks. These benchmarks include fully combinational, scan-based, and fully-sequential circuits. Conclusions and future directions are discussed in

Section 3.4.

3.2 Methodology

This section describes the methodology for test vector evaluation and coverage estimation. This methodology is based on a hypothesis which is used to estimate coverage and evaluate the applied test vector set in a fast manner.

3.2.1 Gate Input Combination Metric

In this section, after discussing a hypothesis, a new metric for coverage estimation and test vector evaluation with a simple example will be discussed.

Hypothesis: For a circuit-under-test, the number of unique input value combinations at each gate input is related to the number of faults detected.

This hypothesis can be described first by giving an example in detail. Suppose we have a circuit containing gate g, which is a 2-input "AND" gate with inputs a and b, and output z. We apply a test vector set containing four test vectors (*tv1*, *tv2*, *tv3*, and *tv4*) to this circuit. If test vector *tv1* causes $\{a, b\} = 01$, *tv2* causes $\{a, b\} = 10$, *tv3* causes $\{a, b\} = 01$ again, and *tv4* does not cause any change in gate g's inputs, then we can say that gate g meets two input combinations out of four possible combinations. In general, we can say that each input combination in each gate can activate new faults on g's inputs and/or propagate the effects of previously activated faults to its output.

In this example, a and b are never set to 1 or 0 simultaneously. For the former case ($\{a, b\} = 11$), *a-stuck-at-0* and *b-stuck-at-0* cannot be **activated**, and for the latter case ($\{a, b\} = 00$), gate g cannot **propagate** the fault effects

from both a and b at the same time with faulty value equal to 1. There are four pairs of data in this example: {(accumulative combinations, accumulative detected faults)} = {(1, 2), (2, 3), (2, 3), (2, 3)}. It can be seen in this example that the growth and saturation of these two numbers follow a similar pattern.

In the above example, the concept of **Gate Input Combination** (GIC) was explained and it showed how GIC can be related to fault coverage. In the real world, the circuits will have many gates, and the number of logic levels of practical circuits is obviously greater than one. There are also many points of reconvergent fanout. For the circuits chosen for this experiment, these two metrics are always highly correlated regardless of the structure of the circuit.

3.2.2 GIC vs. Toggle Count

There is another metric similar to GIC which uses the toggle count in the circuit to evaluate a test vector and estimates the fault coverage [31, 62]. But it should be noted that toggle count is only related to fault activation and thus it can produce a fairly large degree of inaccuracy. This metric is usually used as an upper bound for fault coverage. On the other hand, counting GICs is not only related to fault activation. GICs also represent fault effect propagation in the circuit to some extent. This is because each new gate input combination demands one or more events on its inputs. Recursively, each of these events needs one or more events on its driving gate's inputs. This chain of new events introduces one or more paths with new value combinations which can both activate and propagate the effects of a certain sub-set of faults. However, this does not guarantee that the signals closer to primary outputs are set to values that can propagate the newly generated events to primary

outputs (i.e., non-controlling events). Under certain conditions, like blocking a new event to propagate to primary outputs, can introduce inaccuracy in coverage estimation. I anticipate that considering the gate types and their logic level in the circuit might make the estimation more precise.

3.2.3 Data Correlation

Pearson's correlation coefficient [42] has been used to determine the relationship between GIC and fault detection in a circuit under a set of test vectors. This correlation coefficient indicates the linear correlation between two series of numbers. One number series is the accumulative GIC coverage at each cycle, and the other number series is the accumulative fault coverage at each cycle. Equation 3.1 shows this correlation coefficient. Suppose the two number series (with n members) are named X and Y.

$$\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_x \sigma_y} = \frac{\sum_{i=1}^{n}(X_i - \overline{X}) \times (Y_i - \overline{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \overline{X})^2} \times \sqrt{\sum_{i=1}^{n}(Y_i - \overline{Y})^2}} \tag{3.1}$$

The correlation coefficient is a number between -1 and 1. A value of 0 means "no linear correlation", while value 1 (or -1) means "strong linear correlation". In the case of strong linear correlation, if each corresponding member of the two series represents a point in the Cartesian coordinate system, these points can form a line like in Eq. 3.2.

$$y = \alpha \times x + \beta \tag{3.2}$$

In Eq. 3.2, $x \in Series1$, $y \in Series2$, and α and β are constants. In experiments of this chapter, the correlation coefficients are between 0.84 and 0.998. These coefficients are, for practical purposes, considered "high". Due

Table 3.1: Sample size for various correlation coefficients with power of correlation = 95% and confidence level = 99%

Correlation Coefficient	Sample Size
0.99	6
0.95	8
0.90	11
0.88	12
0.85	14
0.79	18

to [87], when the correlation coefficient is less than 0.35 it is considered "weak correlation", between 0.35 and 0.67 is considered "moderate correlation", and more than 0.68 is considered a "high correlation" coefficient. Coefficients which are more than 0.9 are considered "very high correlation" coefficients [47]. In addition to coefficient values, the size of the series is important, too. For example, if the correlation coefficient between 4 pairs of numbers is 0.99, it cannot be concluded that these numbers are highly correlated even with this high correlation coefficient. To measure how many points are necessary to conclude that the calculated correlation coefficient shows a real correlation (i.e., it does not happen randomly), calculating the power of correlation [42] is required. In this research, a statistical calculator [2] to calculate this factor has been used. Based on this calculator, shown in Tab. 3.1, if there are at least 18 pairs of data, the power of correlation will be 95% for a correlation coefficient equal to 0.79 and confidence level of 99% (i.e., α, which is the chance of making a Type-I error [42], is 0.01). The sample sizes listed in Tab. 3.1 have been used in the experiments (see Tab. 3.2) for this research. These sample sizes cause the power of correlation to be 100% with a confidence level of 99%.

3.2.4 Coverage Estimation and Test Vector Evaluation by GIC

This section discusses how the hypothesis stated in Section 3.2.1 can be utilized for a faster fault grading process. This hypothesis claims that the number of GICs is highly correlated with the number of detected faults in a circuit. Figure 3.1 demonstrates different examples of GIC coverage and fault coverage curves based on the applied test vectors. This figure includes examples of both combinational and sequential circuits. It also shows the results for OR1200 processor (fully-sequential) with two different test vector sets. GIC coverage is defined by Eq. 3.3 and Eq. 3.4.

$$GIC\ Coverage = \frac{\#\ of\ observed\ GICs}{Total\ possible\ GICs} \tag{3.3}$$

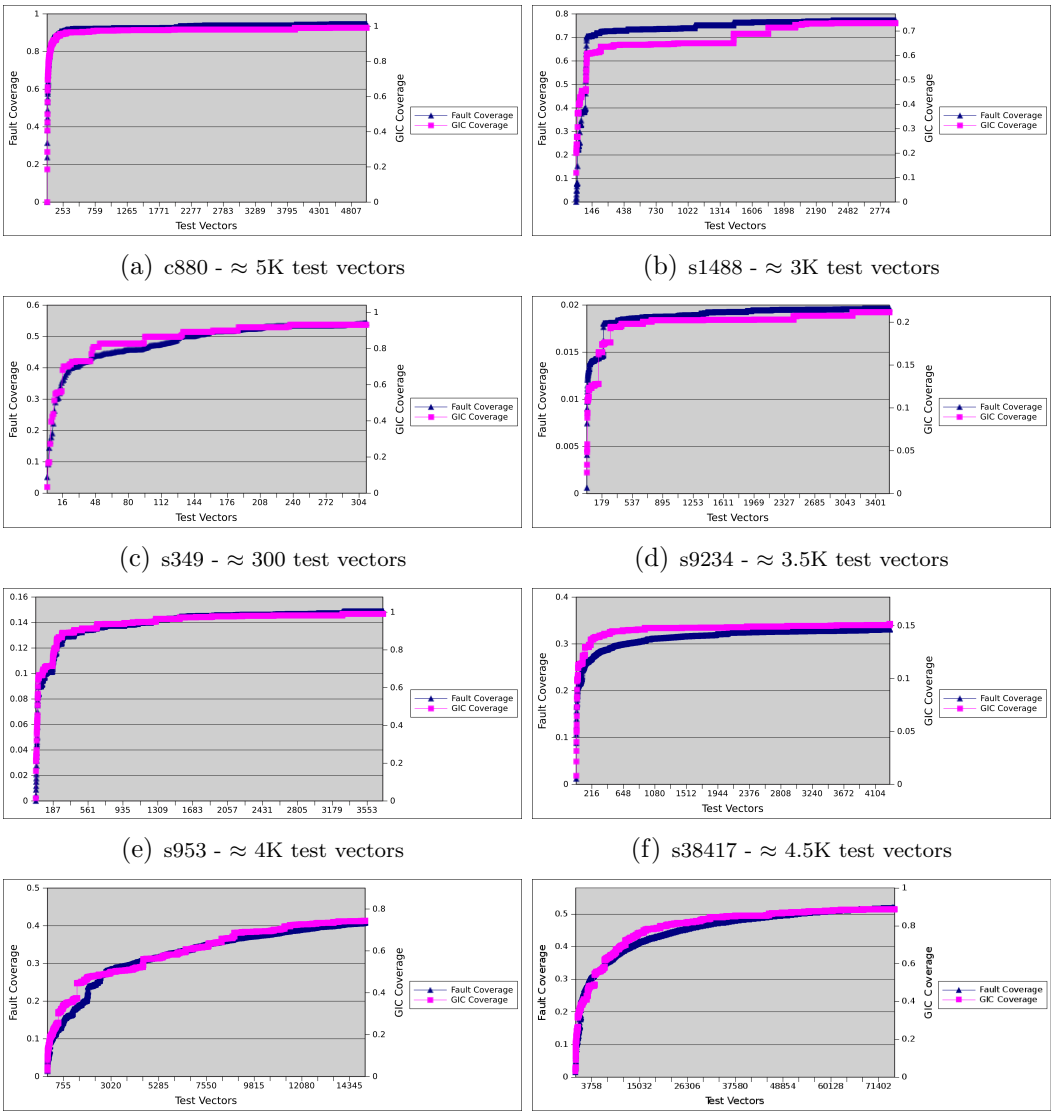
where,

$$Total\ possible\ GICs = \sum_{g \in G} 2^{inputs(g)} \tag{3.4}$$

In Eq. 3.4, G is the set of all gates used in the circuit and *inputs(g)* is the number of inputs for gate g.

As shown in Fig. 3.1, GIC coverage and fault coverage curves grow at the same pace in every case (They both show an exponential nature and then at some point, they start to saturate [95].) The reason for this resemblance is that the two series have a high linear correlation. The more they are correlated, the more their graphs are similar.

Based on this fact, if a circuit is simulated under a set of test vectors and the number of GICs is measured at each cycle, there can be a point where the GIC coverage starts to saturate. Due to the high correlation between GIC and fault coverage, it is very likely that the saturation point is very close to the



(a) c880 - ≈ 5K test vectors

(b) s1488 - ≈ 3K test vectors

(c) s349 - ≈ 300 test vectors

(d) s9234 - ≈ 3.5K test vectors

(e) s953 - ≈ 4K test vectors

(f) s38417 - ≈ 4.5K test vectors

(g) OR1200 - ≈ 15K test vectors

(h) OR1200 - ≈ 75K test vectors

Figure 3.1: GIC and fault coverage curves (vs. simulation cycles) for different circuits and different test vector sets

saturation point in the fault coverage curve. Thus, the following speculations can be made.

1. *Evaluating the test vector set*: If the point of saturation is very early in the simulation, it means that a long time is spent to gain a small coverage. For example, if the applied test vector set contains 1000 vectors, and its fault (GIC) coverage reaches 99% after 100 cycles, this shows that the rest of 900 cycles of simulation is performed to gain only 1% more fault (GIC) coverage. On the other hand, if the coverage grows constantly until the end of the simulation, it means it has not been saturated and if more test vectors are applied, it is very likely that the coverage still grows. In both cases, test engineers need to revise their test vector set. This process needs only one pass of simulation and GIC measurement. Note that the process of test vector evaluation can be repeated several times to reach an effective set of test vectors. Thus, a fast way of evaluating test vectors can save a significant amount of time.

2. *Estimating the fault coverage based on GIC saturation point*: If GIC saturation point on GIC coverage curve is defined as a simulation time like t_s, due to high linear correlation between GIC and fault coverage, it is possible to perform fault simulation until time t_s and, based on this coverage, estimate the fault coverage for the whole test vector set. If t_s is early enough, a considerable amount of time in fault grading process can be saved. Note that estimating coverage is usually tied with test vector set evaluation. So, there might be several test vector sets that should be evaluated and the exact number of fault coverage is not matter in this case.

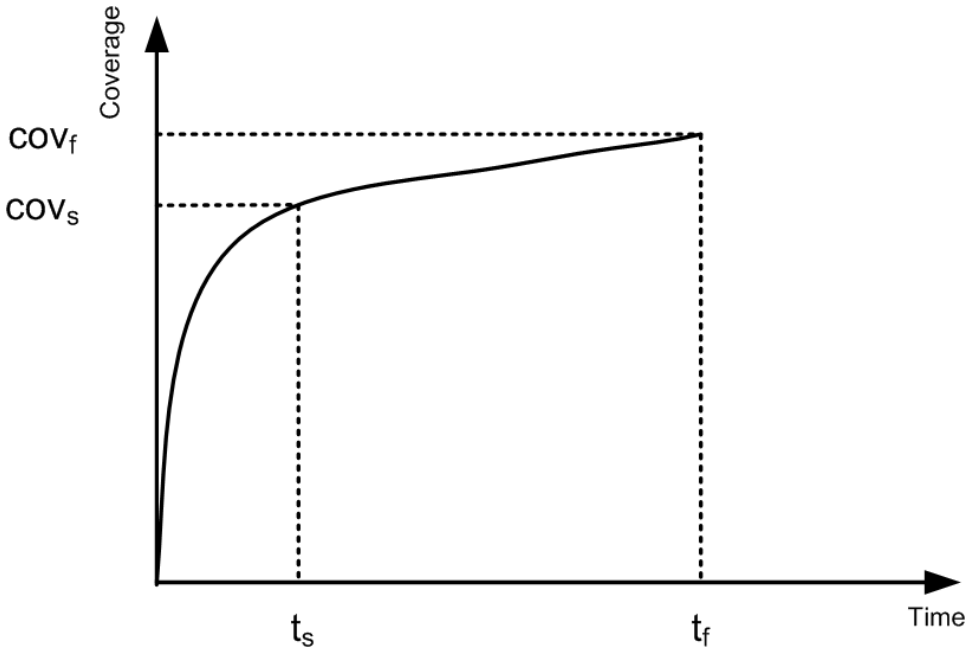


Figure 3.2: Saturation point, (t_s, cov_s), in a typical fault coverage curve

3.2.5 Coverage Saturation Point

Assume a coverage curve like the one in Fig. 3.2. In this figure, t_f and cov_f are the final simulation time and final measured coverage, respectively. I define saturation time (t_s) as the earliest time when the coverage (cov_s) reaches a certain amount of the final coverage (e.g., $cov_s >= 0.99 \times cov_f$). Equation 3.5 finds saturation points as,

$$t_s^{\phi} = min\{t | cov(t) \geq \phi \times cov_f\} \tag{3.5}$$

ϕ can be named as "saturation factor" in Eq. 3.5. I have used t_s^{ϕ} for $\phi = 0.75,\ 0.85,\ and\ 0.95$ in experiments of this chapter (shown in Tab. 3.3).

Based on Eq. 3.5, fault coverage can be calculated based on the saturation point of GIC coverage. Since fault and GIC coverage are linearly

correlated, the pairs (GIC_cov_s, $fault\_cov_s$) and (GIC_cov_f, $fault\_cov_f$) belong to the same line (with small inaccuracy). GIC and fault coverage can be assumed to form a line like the one in Eq. 3.6.

$$fault\_cov = \alpha \times GIC\_cov + \beta \tag{3.6}$$

where, α and β are constants and $\alpha > 0$. The saturation point for GIC coverage is the smallest point where GIC is equal or more than $\phi \times GIC\_cov_f$. Therefore,

$$GIC\_cov_s \geq \phi \times GIC\_cov_f \tag{3.7}$$

Multiplying both sides by α and adding β to them gives,

$$\alpha \times GIC\_cov_s + \beta \geq \alpha \times \phi \times GIC\_cov_f + \beta \tag{3.8}$$

Combining 3.6 and 3.8 results in,

$$fault\_cov_s \geq \phi \times fault\_cov_f + (1 - \phi) \times \beta \tag{3.9}$$

Since at time 0 both GIC and fault coverage numbers are 0, it is expected that β is close to 0. Therefore,

$$fault\_cov_s \geq \phi \times fault\_cov_f \tag{3.10}$$

Equation 3.10 shows that the estimated final fault coverage can be determined using the saturation factor and the calculated saturation point on the GIC coverage curve.

3.2.6 GIC Measurement in Sequential Circuits

This section focuses on GIC measurement in sequential circuits, since it is slightly different from GIC measurement in combinational circuits. In

a sequential circuit, sequential elements like flip-flops contain the state of the circuit and the state value is fed back into the combinational part of the circuit at each clock cycle. Therefore, in fault grading of sequential circuits, there is a chance that a fault is activated in clock cycle t and its effect is propagated through this feedback loop into to the circuit in clock cycles after t. For this reason, sequential elements should be treated in a different way than logic gates in GIC calculation. Since all of these sequential elements hold the state of the circuit, they can be grouped and assumed as a single component. For instance, if there are 10 flip-flops in a circuit which hold the state of the circuit, GIC calculation can assume these flip-flops as one component with 10 inputs and 10 outputs, and as a result with 2^{10} different input combinations. Grouping flip-flops into one virtual component is a heuristic that is effective enough (due to experimental results) for fault coverage estimation. When grouping the flip-flops into one virtual component, the new circuit states, which are the inputs of the flip-flops, generate new GICs and they will give a better estimation for the faults in feedback paths. As another heuristic for calculating GIC coverage in feedback paths, sequential depth of the circuit can be calculated and based on this sequential depth and the number of gates in feedback loops, GIC numbers are counted. This, along with other possible heuristics, can be implemented and tested for sequential circuits in a future experiment.

If each flip-flop is considered individually, it can be considered that there are no fault effects passing through the feedback paths of the sequential circuit. In this case, each flip-flop is basically considered as an output so that there is no fault effect propagation through feedback paths. Therefore, this case is related to full scan-based circuits, since in these circuits, each flip-flop is known as a primary output.

Different sizes of flip-flop groups give different correlation factors. Fortunately, due to experiments, choosing an unsuitable size for flip-flop groups shows itself early in the simulation (the correlation factors degrade from the very beginning of simulation and fault simulation). Therefore, the best group size for flip-flops can be defined intuitively by looking at the first correlation results (e.g., 10 to 100 first cycles). Changing the flip-flop group size does not require re-simulating the circuit. Only a part of GIC calculation regarding flip-flops can be performed again, which has an insignificant effect on run time.

3.2.7 A Special Case

The experimental results in this chapter show that GIC coverage correlates well with fault coverage. However, there are certain situations in which this correlation is not high. In general, the active parts of the circuit aim to generate output values. In some cases, however, some parts of the circuit are active, while their generated events are not propagated through any primary output by the next levels of gates. For example, if some modules in a design are in power-saving mode, their input activities will not be propagated through their outputs. This behavior causes new GICs, but no new faults are detected. Therefore, the correlation between GIC coverage and fault coverage will be degraded. In such designs, control signals such as power management signals or reset signals should be treated carefully. Fortunately, this problem only needs a static analysis of the circuit, and it can only be done once during the design of the test.

In the experiments of Section 3.3, where a global reset signal exists, GICs are not counted when the reset signal is active. The same technique can be applied to power saving signals. Signals dependent on power saving

signals can be identified using circuit analysis; when the power saving signals are active, GICs should not be counted for their dependent signals. This way, the high correlation between GIC and fault coverage is maintained. As an example, the circuit s400 has a special signal called CLR. If this signal is treated as a regular signal, GIC/fault coverage correlation can be degraded to less than 0.5 and/or the estimation inaccuracy can be increased to 16%. When this signal is considered as a reset signal and the GICs are not counted when this signal is active, the correlation between GIC and fault coverage becomes more than 0.9 with low inaccuracy (7.9% and less in the results of this chapter).

3.3 Experimental Results

I have measured the GIC coverage and the fault coverage for ISCAS'85 benchmarks, ISCAS'89 benchmarks, and OR1200 processor [64] for both full-scan and full-sequential (non-scan) modes. In this chapter, to show the efficiency of test vector evaluation by GIC metric, random test vector sets are applied to the circuits. But, in general, GIC is expected to have enough accuracy and run-time performance with ATPG-based test vector sets as well as random test vector sets. Studying the correlation between GIC and fault coverage under set of ATPG-generated test vectors can be considered as a future study. In the full-scan OR1200, flip-flop groups are set as 1 for GIC calculation. All experiments have been performed on a machine with AMD Phenom™ II X6 1055T processor with 0.5 GB of memory. In this experiment, Synopsys VCS® as the logic simulator for calculating GIC coverage numbers and Cadence Verifault-XL® as the fault simulator are used.

As can be seen in the results in this section (Tables 3.2 and 3.3), this

method shows its efficiency when the size of the circuit is moderate to large, and/or the test vector set is large. In such cases, full fault simulation is time consuming and estimating fault coverage using GIC coverage will speed-up the process of test vector evaluation significantly. For large circuits or large test vector sets, the simulation and GIC calculation time is in the order of seconds (minutes in OR1200 case), while the fault simulation run times grow to the order of minutes (hours in OR1200 case).

Table 3.2 shows the run times for both GIC calculation and fault simulation for all benchmarks. Column two of this table shows the number of test vectors applied to the circuit. Columns three and four show the run times for GIC calculation and fault simulation, respectively. The last column shows the correlation coefficient between GIC coverage and fault coverage. This coefficient is measured for the series of GIC and fault coverage numbers per test vector. In these experiments, the correlation coefficients are between 0.84 and 0.998. For data size in these experiments, this range of correlation has a power of correlation equal to 100% and the correlation coefficients are high enough to conclude that the two series of data are (highly) correlated. The last row of Tab. 3.2 is the arithmetic mean of correlation factors and the geometric mean of GIC vs. fault simulation time for the cases with run time greater than 1 minute. As can be seen, the ratio between GIC calculation and fault simulation run times is greater than 11 on average, with an average correlation coefficient of 0.94.

Table 3.3 lists the actual fault coverage achieved by full fault simulation and the inaccuracy in the estimated fault coverage for the benchmarks used in this chapter. Columns two and three in this table show the total number of test vectors applied to each circuit and the actual fault coverage. Since

Table 3.2: Run times for GIC calculation and fault simulation and their correlation coefficients

Design	TV size	GIC Calc. (seconds)	Fault Sim. (seconds)	Correl.
c17	30	0.47	0.02	0.98
c1355	500	0.53	0.24	0.93
c1908	500	0.51	0.55	0.94
c2670	600	0.48	0.66	0.93
c3540	100	0.54	0.24	0.93
c432	360	0.51	0.09	0.97
c499	340	0.46	0.11	0.99
c5315	1600	0.76	1.37	0.89
c6288	500	5	2.44	0.998
c7552	3000	2.31	5.91	0.93
c880	600	0.46	0.23	0.998
s1196	10000	1	4	0.94
s1238	5000	0.84	1.83	0.98
s13207	15000	19	281	0.92
s1488	3000	0.66	1.35	0.997
s15850	14000	23	212	0.96
s27	100	0.46	0.04	0.96
s298	3000	0.48	0.4	0.95
s344	1900	0.56	0.32	0.91
s349	1900	0.48	0.31	0.84
s35932	120	1	8	0.96
s38417	2000	9.08	238.73	0.96
s38584	40000	225.4	2553.1	0.89
s386	1700	0.44	0.26	0.99
s400	5000	0.5	1.4	0.96
s420	950	0.4	0.2	0.98
s444	80	0.4	0.06	0.94
s510	3600	0.52	0.66	0.99
s526	3000	0.5	0.9	0.91
s5378	1750	1.18	3.27	0.99
s641	4500	0.6	1.24	0.99
s713	500	0.5	0.2	0.99
s820	2100	0.62	0.64	0.94
s832	3000	0.5	0.9	0.95
s838	15000	1	6	0.94
s9234	5000	4	45	0.96
s953	4000	0.6	0.6	0.99
OR1200	5000	74	1278	0.99
OR1200	10000	148	2019	0.98
OR1200	30000	477	5017	0.99
OR1200	60000	890	8306	0.99
OR1200	200000	3087	15168	0.94
OR1200(full-scan)	20000	303	1234	0.997
Mean(1 min+)			**11.18x**	**0.96**

the test vector sets are random, some of them are inefficient due to their low fault coverage. In such cases, GIC metric can easily examine these inefficient test vector sets by one pass of fault free simulation. Column five shows the estimated fault coverage inaccuracy based on Eq. 3.5 and Eq. 3.10 (defined in Section 3.2.5) with ϕ equal to 0.75. Column four shows the number of test vectors (and its ratio to the total number of test vectors) that should be applied to reach the saturation point. The inaccuracy in estimated fault coverage is measured based on the following equation.

$$inaccuracy = \frac{|Estimated\ Fault\ Coverage - Simulated\ Fault\ Coverage|}{Simulated\ Fault\ Coverage} \tag{3.11}$$

The next columns measure the same parameters as columns four and five, but with different values for ϕ. As shown in Tab. 3.3, a larger ϕ usually causes less inaccuracy in estimated coverage but more test vectors to fault simulate are needed. The last row in this table shows the range for the test vector rates and inaccuracies for each ϕ. In this table, to have more precise results, bootstrapping method (with size of 1000) has been used to find these ranges. The confidence level used in this experience is 95%.

One important fact in Tab. 3.3 is that the inaccuracy (error) varies over a wide range in the experimental results. This error depends on the distance of the chosen saturation point to the regression line between GIC and fault coverage drawn up to that point. In the next chapter, a solution for defining an error bound using regression method.

In order to show how much run time can be saved with this simple estimation method in large circuits, fault simulation run time for the estimated test vector sizes in Tab. 3.3 for $\phi = 0.75$ is shown (Tab. 3.4). In this table, on

Table 3.3: Estimated and measured fault coverage with different saturation factors

Design	TV size	Fault Cov.(%)	TV size/Ratio (ϕ=0.75)	Error (%) (ϕ=0.75)	TV size/Ratio (ϕ=0.85)	Error (%) (ϕ=0.85)	TV size/Ratio (ϕ=0.95)	Error (%) (ϕ=0.95)
c17	30	100	12 / 40%	0	13 / 43.33%	0	14 / 46.66%	0
c1355	500	96.2	24 / 4.8%	9.3	39 / 7.8%	9.4	179 / 35.8%	0.2
c1908	500	91.6	14 / 2.8%	10.3	35 / 7%	9.8	100 / 20%	7.9
c2670	600	82	14 / 2.33%	7.8	21 / 3.5%	7.5	48 / 8%	6.7
c3540	100	80.4	22 / 22%	12.4	30 / 30%	10.9	46 / 46%	9.4
c432	360	96.6	24 / 6.66%	9.8	28 / 7.77%	8.3	88 / 24.44%	0.02
c499	340	97.6	21 / 6.17%	2.3	36 / 10.58%	1.6	130 / 38.23%	0.49
c5315	1600	99.2	18 / 1.12%	9.7	25 / 1.56%	9.7	42 / 2.62%	9.5
c6288	500	99.4	9 / 1.8%	0.53	11 / 2.2%	0.53	16 / 3.2%	0.5
c7552	3000	93.7	14 / 0.46%	10.7	20 / 0.66%	8.8	38 / 1.26%	8.7
c880	600	97.3	15 / 2.5%	5	22 / 3.66%	4.6	58 / 9.66%	3
s1196	10000	95.1	373 / 3.73%	4.2	1290 / 12.9%	5.1	4887 / 48.87%	1.8
s1238	5000	86.1	208 / 4.16%	10.8	588 / 11.76%	0.7	2170 / 43.4%	0.2
s13207	15000	38.3	548 / 3.65%	5.4	2369 / 15.79%	4.4	8494 / 56.62%	0.2
s1488	3000	70.2	678 / 22.6%	0.59	682 / 22.73%	0.61	1275 / 42.5%	0.69
s15850	14000	46.6	315 / 2.25%	3.8	950 / 6.78%	3.3	5066 / 36.18%	1.2
s27	100	98	12 / 12%	2	17 / 17%	6	34 / 34%	2
s298	3000	83	512 / 17.06%	8.5	1251 / 41.7%	2.1	1515 / 50.5%	1.9
s344	1900	97.5	249 / 13.10%	2.4	771 / 40.57%	2.4	1493 / 78.57%	2.4
s349	1900	96.2	11 / 0.57%	5.2	29 / 1.52%	2	292 / 15.36%	3.4
s35932	120	86.8	21 / 17.5%	9.5	58 / 48.33%	9.7	86 / 71.66%	5
s38417	2000	15.3	19 / 0.95%	10.8	54 / 2.7%	10.3	175 / 8.75%	5.8
s38584	40000	68.1	170 / 0.42%	10.9	501 / 1.25%	9.8	1986 / 4.96%	8.9
s386	1700	65.3	125 / 7.35%	5.1	382 / 22.47%	4.3	527 / 31%	0.8
s400	5000	88.2	88 / 1.76%	7.9	123 / 2.46%	6.7	195 / 3.9%	1.8
s420	950	61.9	292 / 30.73%	1.2	425 / 44.73%	1.7	543 / 57.15%	2.7
s444	80	16.6	20 / 25%	7.6	23 / 28.75%	7	47 / 58.75%	2.8
s510	3600	100	49 / 1.36%	8.7	60 / 1.66%	6.3	86 / 2.38%	4.8
s526	3000	11.9	704 / 23.46%	8.3	706 / 23.53%	7.3	1199 / 39.96%	3.9
s5378	1750	65.2	16 / 0.91%	3.8	54 / 3.08%	4.1	489 / 27.94%	0.4
s641	4500	87.9	14 / 0.31%	3.3	46 / 1.02%	0.6	476 / 10.57%	7.7
s713	500	82.6	39 / 7.8%	7.2	56 / 11.2%	4.9	198 / 39.6%	0.2
s820	2100	43.4	61 / 2.90%	6	221 / 10.52%	7.8	417 / 19.85%	0.41
s832	3000	49.1	158 / 5.26%	9.6	243 / 8.1%	7.7	717 / 23.9%	1.6
s838	15000	45.7	244 / 1.62%	10.5	2010 / 13.4%	10.9	6180 / 41.2%	0.5
s9234	5000	20.8	144 / 2.88%	3.3	146 / 2.92%	0.1	1256 / 25.12%	1.8
s953	4000	99.2	99 / 2.47%	0.7	276 / 6.9%	0.7	1181 / 29.52%	0.7
OR1200	5000	50.5	1929 / 38.58%	0.3	2941 / 58.82%	7.6	4379 / 87.58%	1.5
OR1200	10000	67.7	2566 / 25.66%	4.6	4248 / 42.48%	3.1	7659 / 76.59%	1.7
OR1200	30000	83.2	6383 / 21.27%	4	11924 / 39.74%	1.7	21910 / 73.03%	1.6
OR1200	60000	88.1	9821 / 16.36%	1.4	20200 / 33.66%	1.5	39722 / 66.20%	0.4
OR1200	200000	92.6	18362 / 9.18%	1	39402 / 19.70%	2.4	111031 / 55.51%	1.9
OR1200 (full-scan)	20000	89.6	3295 / 16.47%	2.6	5086 / 25.43%	0.3	10811 / 54.05%	0.2
G.Mean (bootstrap)				**3.64 ± 0.05**		**2.84 ± 0.04**		**1.22 ± 0.02**

average, fault grading is 8.25 times faster by sacrificing 2.85% coverage accuracy. If we consider the number of test iterations before tape out (which can be around 4 iterations), the difference between full and partial fault simulation run times becomes even more. In general, run time speedup depends on factors like ϕ and it also depends on how effective is the test vector set in detecting faults. In larger circuits with a large set of test vectors that might detect a small percentage of faults, the run time speed up between the estimation in

Table 3.4: Run times for fault simulation based on saturation estimation

Design	Original/Reduced TV size	Original Fault Sim.(sec)	Partial Fault Sim.(sec)	Fault Sim. Speedup
OR1200	5000 / 1929	1278	516	2.47
OR1200	10000 / 2566	2019	795	2.53
OR1200	30000 / 6383	5017	1642	3.05
OR1200	60000 / 9821	8306	2714	3.06
OR1200	200000 / 18362	15168	3584	4.23
OR1200 (full-scan)	20000 / 3295	1234	228	5.41
s13207	15000 / 548	195	8	21.96
s38417	2000 / 19	238	3	69.19
s38584	40000 / 170	2553	29	87.22

this chapter and fault simulation will get even more significant.

3.4 Conclusions and Future Directions

A statistical metric based on gate input combinations was introduced in this chapter. Experiments on ISCAS'85, ISCAS'89, and the OR1200 CPU model show that the introduced metric has a high linear correlation with fault coverage in combinational and sequential circuits. This correlation can be used for test vector set evaluation and fault coverage estimation. An average inaccuracy of 3.67% in estimated fault coverage by this metric was observed, by only fault simulating 4.3% of the test vectors (on average). In the next chapter, this metric is used to estimated fault coverage with lower and upper bounds. This method can be useful for all types of circuits, but it shows its effectiveness in large sequential circuits which need a large sequence of test vectors to reach an acceptable fault coverage.

It is expected that the introduced metric can be even more accurate if

the type of gate and its logic level in the circuit are taken into consideration. This can be tested in future implementations. Also, for a more precise GIC calculation in feedback paths, GIC counting based on the sequential depth of the circuit can be implemented and analyzed for different sequential circuits. Another future direction can be to relate GICs to faults due to their position in the design. If there is such a relationship, GIC will have the ability to locate the parts of the circuit which could not be tested efficiently with the applied test vector set. This analysis can also lead to a hybrid structural test and GIC estimation.

Chapter 4

A Regression Model for Fault Coverage Estimation using GIC Metric

4.1 Overview

Regression-based methods [14] are widely used in economics and medical experiments to predict the future outcomes of a system. In this chapter, simple linear regression is employed to estimate the fault coverage of tests for a system. This methodology is named EAGLE (linEAr reGression-based fauLt coverage Estimation) [52]. This method takes advantage of a strong linear relationship between GIC coverage and fault coverage (see Chapter 3). Based on this relationship, EAGLE uses partial fault simulation (for the first n test vectors) to fit a line using simple linear regression model. According to this line and the values of the GIC metric, which are calculated by only one pass of simulation, fault coverage is estimated within a range for the whole test vector set. Experimental results in this chapter show that, on average, by fault simulation of 7.6% of test vectors, the final fault coverage is correctly estimated for 94% of the cases. This regression based estimation model has been tested for 60 different configurations on 33 different circuits, each with 50 different test vector sets. This means that the results in this chapter are based on 99,000 different cases. The main contribution of this chapter is building a system with different parameters for estimating the fault coverage within a range.

The rest of this chapter is organized as follows. Section 4.2 explains simple linear regression and its limitations. In Section 4.3, EAGLE algorithm and its acceptable constraints are discussed. In Section 4.4, it is shown that how the correctness of the estimations changes by modifying the EAGLE constraints. Section 4.5 contains the conclusions.

4.2 Simple Linear Regression

In statistics [14], when we have a set of data (x_i, y_i), to estimate y by having the x value, we can fit a line between a group of available data points and estimate y, named $\hat{y}$, using that fitted line. The act of calculating the "best fit" line for a set of data points is called linear regression. We can always find the best fit line for any set of data. However, a meaningful estimation of y demands x_i and y_i values to have a fairly high linear correlation.

In simple linear regression, y_i values are related to only one set of values (e.g., x_i), while in multiple linear regression, y_i can have more than one predictor (e.g., x_{1_i} to x_{p_i}). In simple linear regression, the best fit line is calculated in a way that the sum of squared residuals is minimized.

Suppose it is needed to pass a line through n data points, (x_1, y_1) to (x_n, y_n). For these n data points, simple linear regression algorithm finds β_0 and β_1 such that:

$$\hat{y}_i = \beta_1.x_i + \beta_0 \qquad (i \in \{1..n\}) \tag{4.1}$$

Residual of the i^{th} point is defined as the difference between the real value of y_i and the estimated value $\hat{y}_i$ using the regression line:

$$r_i = y_i - \hat{y}_i \tag{4.2}$$

Therefore, the best fit happens when $\Sigma {r_i}^2$ is minimized. This method is also called the simple ordinary least squares (OLS). As shown in Equations 4.3 and 4.4, β_1 and β_0 can be calculated as,

$$\beta_1 = r_{xy}.\frac{\sigma_y}{\sigma_x} \tag{4.3}$$

$$\beta_0 = \overline{y} - \beta_1.\overline{x} \tag{4.4}$$

where r_{xy} is the correlation coefficient between x and y values, σ_x and σ_y are the standard deviation of x_i and y_i values, respectively, and $\overline{x}$ and $\overline{y}$ are the mean of x_i and y_i values, respectively.

A regression line is used to estimate y values for some given x values. A metric that determines how many of the points used in the regression model are close enough to the regression line, is the squared of correlation coefficient between x_i and y_i values (${r_{xy}}^2$). Therefore, this estimation model is useful only when we know that the set of x and y values have a high linear correlation coefficient. Otherwise, there is no guarantee that the estimated value is close to the actual value. In experiences shown in this chapter, with correlation factor values between 0.87 and 0.998, r^2 factor will be between 0.756 and 0.996.

The detailed discusstion in Chapter 3 showed that the cumulative GIC coverage and the cumulative stuck-at fault coverage numbers have a strong linear correlation. This means that if there are n points, each consisting of a GIC coverage value for a set of test vectors and its corresponding fault

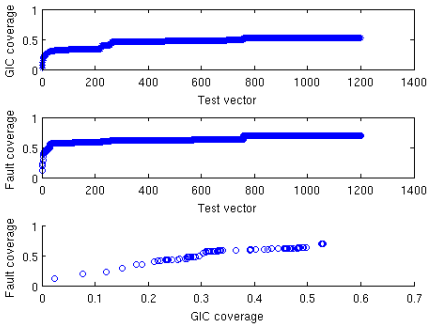
coverage value, say, (gic_1, fc_1) to (gic_n, fc_n) and the best fit line for these n points is calculated, fc_k by having gic_k and the regression line $(k > n)$ can be estimated. Figure 4.1 shows this relationship by plotting the GIC coverage vs. the number of applied test vectors, fault coverage vs. the number of applied test vectors, and fault coverage vs. GIC coverage in the upper, middle, and lower curves in each part, respectively. As can be seen in this figure, the curves for different test vectors look different, but GIC and fault coverage curves are always very similar and the lower curve is always very close to a line.

Since the regression line between each subset of data points can be different (unless all points are exactly on the same line), there should be a way to find a range of regression lines instead of only one line for a set of data points. Due to [14], the *estimation interval* for fc_0 based on the value of gic_0 can be calculated as,

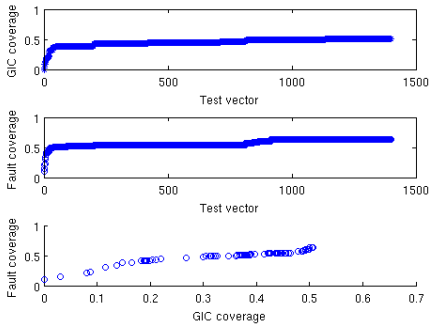
$$\hat{fc_0} - t_{\alpha/2}^{n-2} \hat{se}.(\hat{fc_0}) \leq fc_0 \leq \hat{fc_0} + t_{\alpha/2}^{n-2} \hat{se}.(\hat{fc_0}) \tag{4.5}$$

where,

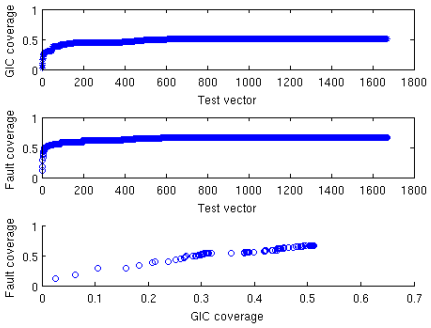
- n is the number of data points used in regression.
- $\hat{fc_0}$ is the calculated fault coverage based on gic_0 value ($\hat{fc_0} = \beta_1.gic_0 + \beta_0$). In this chapter, to predict the final fault coverage, final GIC number (GIC number at the end of simulation) is used as gic_0.
- $t_{\alpha/2}^{n-2}$ is the t-value (from t-distribution) with confidence level of $(1-\alpha)\%$ and $n-2$ degrees of freedom.
- $\hat{se}.(\hat{fc_0})$ is the estimated standard error of the estimated fault coverage based on gic_0 value. Equations 4.6, 4.7, and 4.8 express this term.



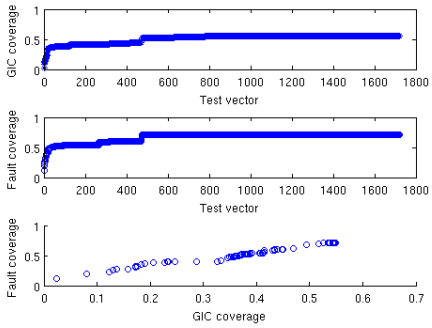
(a) $r_{gic,fc} = 0.90$



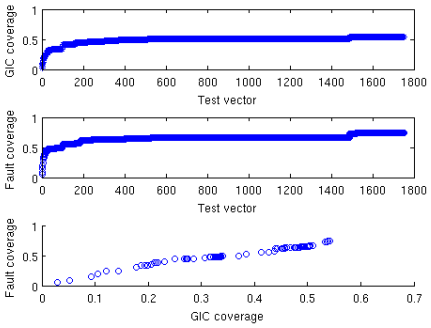
(b) $r_{gic,fc} = 0.91$



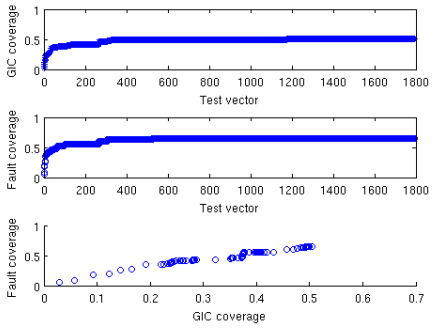
(c) $r_{gic,fc} = 0.97$



(d) $r_{gic,fc} = 0.99$



(e) $r_{gic,fc} = 0.97$



(f) $r_{gic,fc} = 0.98$

Figure 4.1: GIC and fault coverage curves in s386 circuit for 6 different test vector sets. In each subfigure, GIC coverage vs. the number of simulated test vectors (upper curve), fault coverage vs. the number of simulated test vectors (middle curve), and the scatter plot for fault and GIC coverage (lower curve) are shown.

$\hat{s.e.}(\hat{fc_0})$ shows the quality of the regression line passed through the data points and the distance of the final GIC number to the GIC numbers in the data points.

$$\hat{s.e.}(\hat{fc_0}) = \hat{\sigma} \sqrt{1 + \frac{1}{n} + \frac{(gic_0 - \overline{gic})^2}{\sum_{i=1}^{n}(gic_i - \overline{gic})^2}} \tag{4.6}$$

$$\hat{\sigma}^2 = \frac{\sum_{i=1}^{n}(fc_i - \hat{fc_i})^2}{n-2} \quad (residual\ mean\ square) \tag{4.7}$$

$$\overline{gic} = \frac{\sum_{i=1}^{n} gic_i}{n} \tag{4.8}$$

The formulae discussed above show that if there is a set of data points, and a regression line is calculated based on a subset of these points, the estimated value for each subset might be different, but it will be likely within a range of values. This way, the final fault coverage within a range can be estimated, if the final GIC coverage for a set of test vectors is available. In this case, there is a need to have fc_i and gic_i values for a number of test vectors (the first n test vectors) and the final GIC value. One pass of good (i.e., fault-free) simulation to retrieve all the GIC values and fault simulation for the first n test vectors to obtain n fc values.

Although the statistical formulae above are used for calculating the range of estimated fault coverage, there are some pre-conditions that should hold for the data points so that the above equations can be used. If any of these conditions does not hold, there is a chance to observe unexpected results. These conditions are listed below.

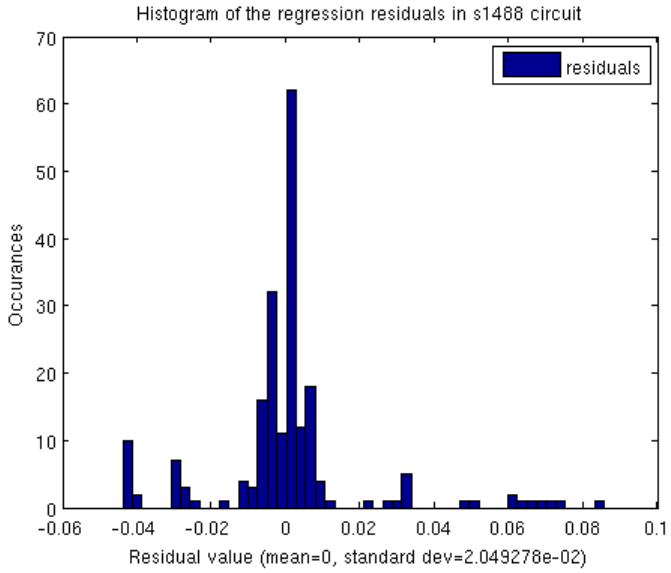


Figure 4.2: A sample histogram of regression residuals

1. For each set of data points, the mean of the residuals should be zero.
2. For each set of data points, the variance of residuals should be constant.
3. The residuals should be uncorrelated with each other.
4. The residuals should be normally distributed.

Again, "residual" means the difference between each estimated fault coverage and the real fault coverage.

The data in the experimental results of this chapter were tested for the conditions above[1]. Conditions 1 and 2 are met. For condition 3 (a.k.a. auto-correlation), the scatter plots of the residuals against GIC values were sketched

[1]Note that the conditions are not proved to be always correct. They should be checked for each new design before using the mentioned formulae.

and they did not show a pattern. Therefore, it can be assumed that condition 3 is also met for these experiments. Condition 4 does not hold perfectly in these experiments. This means that the histogram for residuals is not a perfect symmetric bell-shaped histogram. However, the histograms follow the pattern of having almost 95% of data between 2σ and around 99.7% of data between 3σ from the mean of residuals, where σ is the standard deviation of the residuals. This way, the formulae above can be still used, but it is important to note that using confidence levels less than 95% is not safe and might give misleading results. Due to the observations done for this chapter's experiments, when the number of test vectors and the circuit size grow, the histograms become closer to bell-shaped histograms, meaning that it is safe to use this regression model for large sets of test vectors in large circuits. Figure 4.2 shows the residual histogram in circuit s1488 as an example.

In the next section, it is shown how to use the linear regression model, discussed in this section, to estimate the value of fault coverage under application of a test vector sequence.

4.3 EAGLE Steps

As discussed in Section 4.2, a linear regression model can be built based on a partial fault simulation and the GIC coverage.

If a desired range for the estimated coverage and a confidence level (95% or more) are given by the user, EAGLE can iteratively perform fault simulation, build a regression model, and calculate the range of estimated fault coverage until it reaches a point that the calculated range is equal (or less than) the desired range. Since the regression model might never converge to the given range, to save time, EAGLE should stop its iterations when reaching

a certain number of test vectors and inform the user that the algorithm has not converged yet with that number of test vectors and output the latest calculated range. The user can decide to increase the number of test vectors for partial fault simulation and **resume** the estimation algorithm, or the given range is acceptable anyways (i.e., it does not have a big difference with the desired range). The flowchart of this algorithm is shown in Fig. 4.3.

The constraints that can be changed for each estimation are listed below.

- ϵ: The number of test vectors that are fault simulated in each iteration of algorithm. For example, I have set ϵ equal to 5 in this chapter's experiments, meaning that 5 test vectors are applied to the circuit by fault simulator before each regression line calculation.

- **Confidence level**: The confidence level that is used to determine the range of estimated coverage. For example, if confidence level is 95%, it means that the probability of the real coverage being in the estimated range is 95%. However, since the residuals do not follow a perfect normal distribution, this probability might be less or more than the probability that the confidence level determines.

- **range**: The user can define a desired coverage range. This can be the goal of the regression model. For example, the range of 5% means that the user prefers that the lower and upper bounds of the estimated coverage do not differ more than 5% (e.g., $fc \in [40\%\ 45\%]$).

- **Test vector limit**: The cutoff point for the algorithm in case that the given range cannot be reached. Remember that the goal is to rapidly

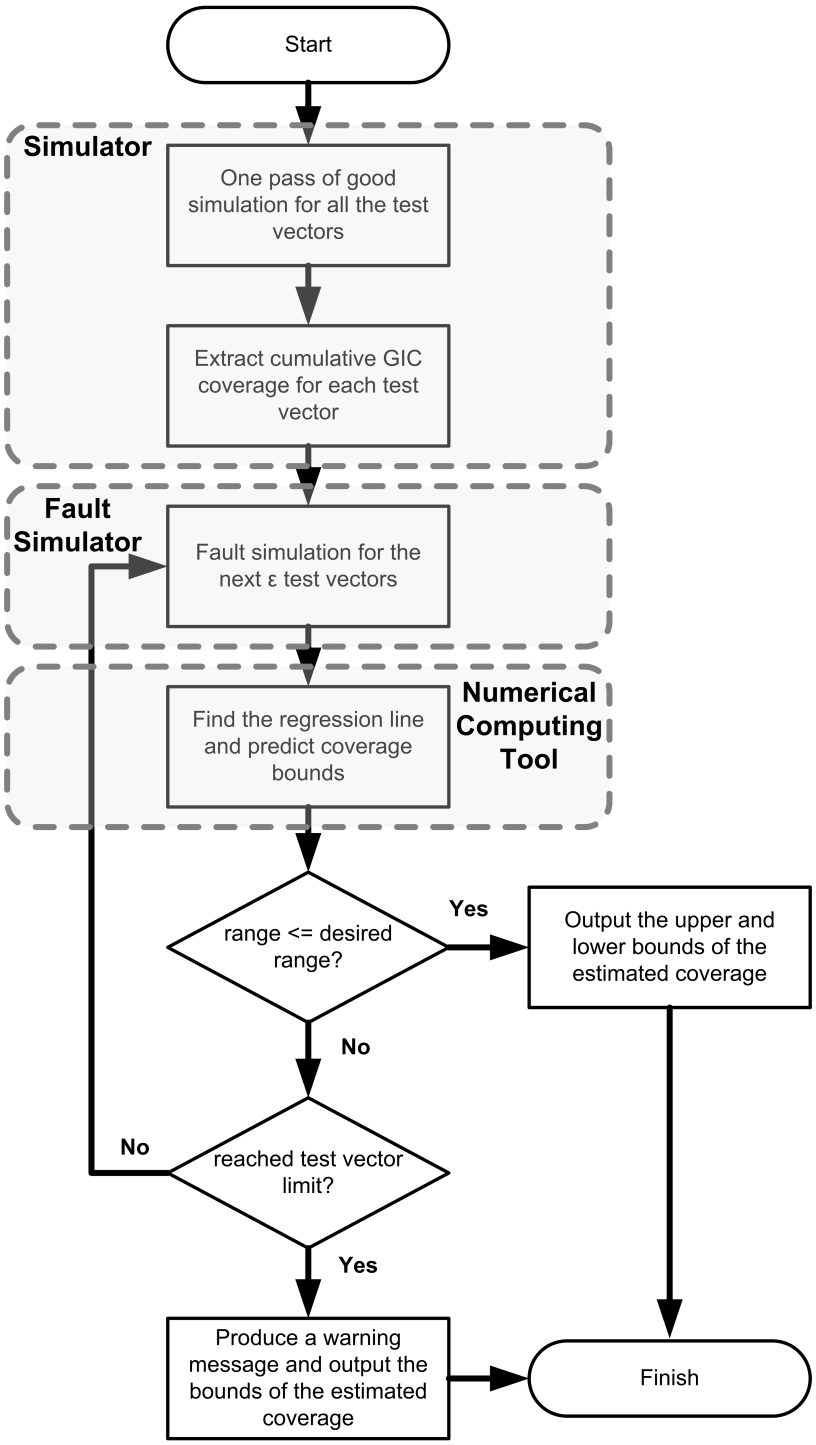


Figure 4.3: EAGLE flowchart

estimate the coverage. Therefore, it is not desirable to fault simulate, for example, 80% of the test vectors to reach a small range of estimated coverage. However, the user can set large limits at first or at each stop point and resume fault simulation and estimation process from that stop point.

EAGLE has been implemented and used for estimating fault coverage for various circuits and test vector sets. The results of these estimations will be discussed in the next section.

4.4 Experimental Results

EAGLE has been applied to 32 circuits belonging to ISCAS'85 and ISCAS'89 benchmarks, and OR1200 processor [64]. Table 4.1 shows some statistics for these circuits. The tools that have been used in EAGLE are Synopsys VCS® as the simulator, Cadence Verifault-XL® as the fault simulator, and Matlab® as the numerical computing tool to calculate regression lines and coverage bounds.

To obtain more accurate statistical results, for each circuit, 50 experiments have been performed with different sets of random test vectors (with different number of test vectors in each set). Then for each experiment, a regression model for different constraint settings has been built. In this chapter, the maximum number of test vectors that EAGLE uses to fault simulate, the confidence level, and the desired range of estimated coverage are the constraints that can be changed by the user before estimating the coverage by EAGLE. Different values are given to these parameters for each configuration. These values are shown in Tab. 4.2. These values result in 60 different

Table 4.1: Benchmark statistics used for EAGLE evaluation. Numbers in percentage format show the ratio of the number of gates in a group (e.g., number of AND gates) to the total components (×100).

Circuit name	Total	AND (%)	BUF (%)	DFF (%)	NAND (%)	NOR (%)	NOT (%)	OR (%)	XNOR /XOR (%)	1-inp	2-inp	3-inp	4^{+}-inp
c1355	554	11.6	0.0	0.0	75.1	0.0	13.0	0.4	0.0	13.0	82.3	2.9	1.8
c1908	956	8.9	16.9	0.0	42.3	0.1	29.0	2.8	0.0	45.9	45.2	7.2	1.7
c2670	1278	26.6	21.3	0.0	19.9	0.9	25.1	6.2	0.0	46.4	40.7	10.3	2.6
c3540	1703	30.3	13.1	0.0	17.5	4.9	28.8	5.4	0.0	41.9	44.6	10.6	2.9
c432	168	7.1	0.0	0.0	47.0	11.3	23.8	0.0	10.7	23.8	60.7	2.4	13.1
c499	202	27.7	0.0	0.0	0.0	0.0	19.8	1.0	51.5	19.8	71.3	0.0	8.9
c5315	2330	31.5	13.4	0.0	19.5	1.2	24.9	9.5	0.0	38.4	38.1	19.5	4.0
c6288	2416	10.6	0.0	0.0	0.0	88.1	1.3	0.0	0.0	1.3	98.7	0.0	0.0
c7552	3569	22.6	15.0	0.0	28.8	1.5	24.5	7.5	0.0	39.5	49.9	7.8	2.7
c880a	383	30.5	6.8	0.0	22.7	15.9	16.4	7.6	0.0	23.2	66.6	6.8	3.4
s1196a	547	21.6	0.0	3.3	21.8	9.1	25.8	18.5	0.0	25.8	55.6	13.9	1.5
s1238	526	25.5	0.0	3.4	23.8	10.8	15.2	21.3	0.0	15.2	63.1	16.5	1.7
s13207	8589	13.0	0.0	7.4	9.9	1.1	62.6	6.0	0.0	62.6	25.4	1.7	2.9
s1488	659	53.1	0.0	0.9	0.0	0.0	15.6	30.3	0.0	15.6	60.7	17.6	5.2
s344	175	25.1	0.0	8.6	10.3	17.1	33.7	5.1	0.0	33.7	53.1	4.6	0.0
s349	176	25.0	0.0	8.5	10.8	17.6	32.4	5.7	0.0	32.4	54.5	4.5	0.0
s382	179	6.1	0.0	11.7	16.8	19.0	33.0	13.4	0.0	33.0	34.1	16.2	5.0
s38417	23815	17.4	0.0	6.9	8.6	9.6	56.6	0.9	0.0	56.6	32.4	3.6	0.6
s38584	20679	26.7	0.0	6.9	10.3	5.7	37.7	12.7	0.0	37.7	48.7	3.3	3.3
s386	165	50.3	0.0	3.6	0.0	0.0	24.8	21.2	0.0	24.8	37.6	25.5	8.5
s400	184	6.0	0.0	11.4	19.6	18.5	31.0	13.6	0.0	31.0	35.3	17.4	4.9
s420	234	20.9	0.0	6.8	12.4	14.5	33.3	12.0	0.0	33.3	50.4	8.1	1.3
s444	207	6.3	0.0	10.1	28.0	16.4	31.4	6.8	0.0	31.4	38.6	14.5	5.3
s510	222	15.3	0.0	2.7	27.5	24.8	15.8	13.1	0.0	15.8	68.0	11.7	1.8
s526	214	26.2	0.0	9.8	10.3	16.4	24.3	13.1	0.0	24.3	31.8	16.4	17.8
s5378	2958	0.0	0.0	6.1	0.0	25.9	60.0	8.1	0.0	60.0	21.5	10.3	2.1
s641	398	22.6	0.0	4.8	1.0	0.0	68.3	3.3	0.0	68.3	16.8	6.8	3.3
s713	412	22.8	0.0	4.6	6.8	0.0	61.7	4.1	0.0	61.7	22.6	8.0	3.2
s832	292	26.7	0.0	1.7	18.5	22.6	8.6	21.9	0.0	8.6	41.8	20.5	27.4
s838	478	22.0	0.0	6.7	11.9	14.6	33.1	11.7	0.0	33.1	50.6	8.2	1.5
s9234	5808	16.4	0.0	3.6	9.1	1.9	61.5	7.4	0.0	61.5	31.1	1.6	2.2
s953	424	11.6	0.0	6.8	26.9	26.4	19.8	8.5	0.0	19.8	65.1	7.8	0.5
OR1200	35580	4.7	9.1	5.2	48.2	9.0	14.9	1.7	7.1	24.0	64.7	6.1	0.0

configurations that can be tested on each circuit for each set of test vectors.

From Tab. 4.2, there are 60 configurations which end up running EAGLE for 99,000 cases ($33 \times 50 \times 60$), and specify if the estimated coverage matches the real coverage. For each case, the outcomes listed below are calculated and reported. The acronyms used in tables and figures are also shown below in parentheses.

- The percentage of *all* EAGLE estimations whose ranges were too wide (**wide%**).

[1] As discussed in Section 4.2, using confidence levels less than 95% is not recommended in this regression model.

Table 4.2: Given values for EAGLE constraints

Constraint	Tested Values
ϵ	5
range	5%, 7%, 10%, 15%
test vector limit	10%, 20%, 30%, 40%, 50%
confidence level[1]	95%, 98%, 99.8%

- Among the cases whose ranges were converged to a narrow enough range, the percentage of the cases that the real fault coverage lies in the estimated coverage range (**correct$_{\text{converged}}$%**).
- Among *all* cases, the percentage of the cases that the real fault coverage lies in the estimated coverage range (**correct$_{\text{all}}$%**).
- Among *all* the estimations, the percentage of estimations that are within range or have at most 3% coverage difference with the estimated range (**diff3%**).
- Among *all* the estimations, the percentage of estimations that are within range or have at most 5% coverage difference with the estimated range (**diff5%**).
- The percentage of test vectors that are used for partial fault simulation (**tv%**).
- The percentage of *all* cases that reach the maximum test vector limit defined in their configurations before converging to the desired range (**tv-lim%**).

As discussed in Section 4.3, the algorithm might not converge to the desired range before the test vector limit is reached. In such cases, the range

calculated by the last iteration is given to the user. In some cases, this range is so wide that it is not useful for the user at all. For example, if the user needs to know the fault coverage within no more than 2% (i.e., the desired range is 4%), and the defined maximum test vectors are not sufficient for reaching the 4% range, the algorithm exits and outputs its latest calculated range. If this range is 20%, it is very unlikely that the user, who wants a 4% range, can take any advantage of this wide range. However, if the result range is around 6%, it might be still helpful to the user. In EAGLE experiments, the result of an estimation is considered as too wide if the estimated range is 5% more than the desired range specified in the configuration. For example, if the desired coverage range is 10%, the estimations that result in a range equal or more than 15% are assumed too wide. However, the definition of having a wide range totally depends on the user.

Based on the above assumptions, the outcomes, such as the percentage of wide range cases, the percentage of correct estimations, the percentage of test vectors that are used for partial fault simulation, and the percentage of the cases that reached the user-defined test vector limit without reaching the user-defined range constraint are calculated and shown in the following tables. These outcomes (averaged over configurations) are shown for each circuit in Tab. 4.3. In this table, the amount of wide range, the percentage of correct estimation among converged ranges and among all cases have been shown. Note that $correct_{converged}\%$ numbers are the percentage of correct estimations that are **not** marked as wide (their estimated range was around the user's desirable range). $correct_all\%$ numbers are the percentage of correct estimations among all cases. Also, *diff3%* and *diff5%* are the percentage of correct estimations (with up to 3% or 5% error) among **all** the estimations.

Table 4.3: EAGLE estimation outcomes based on each circuit (averaged over configurations)

Circuit	wide%	$correct_{converged}$%	$correct_{all}$	diff3%	diff5%	tv%	tv-lim%
c1355	3.40%	100.00%	96.6%	98.78%	100.00%	18.95%	47.11%
c1908	50.26%	72.97%	36.3%	82.48%	91.63%	24.96%	78.48%
c2670	0.48%	98.22%	97.7%	99.35%	100.00%	9.24%	13.03%
c3540	66.37%	96.73%	32.5%	43.53%	96.30%	28.38%	90.77%
c432	0.78%	100.00%	99.2%	99.86%	100.00%	21.76%	61.90%
c499	1.12%	100.00%	98.9%	99.96%	100.00%	25.41%	74.78%
c5315	59.67%	97.68%	39.4%	92.15%	100.00%	28.29%	91.14%
c6288	0.03%	100.00%	100%	100.00%	100.00%	12.45%	20.62%
c7552	18.15%	99.91%	81.8%	85.40%	93.62%	20.19%	53.59%
c880a	0.44%	100.00%	99.6%	99.63%	100.00%	8.41%	8.78%
s1196a	6.56%	92.49%	86.4%	92.43%	94.90%	26.40%	83.40%
s1238	56.49%	85.68%	37.3%	50.72%	60.62%	27.62%	87.14%
s13207	0.00%	97.30%	97.3%	99.90%	100.00%	5.99%	2.13%
s1488	28.37%	85.84%	61.5%	75.67%	85.04%	20.14%	55.60%
s344	6.96%	99.72%	92.8%	95.85%	98.96%	28.35%	90.63%
s349	3.41%	99.31%	95.9%	96.44%	97.07%	28.97%	95.30%
s382	0.00%	100.00%	100%	100.00%	100.00%	5.01%	0.00%
s38417	0.17%	99.83%	99.7%	99.83%	99.90%	6.17%	2.33%
s38584	0.20%	100.00%	99.8%	99.80%	99.87%	5.41%	1.07%
s386	9.65%	87.13%	78.7%	87.33%	91.56%	15.62%	36.04%
s400	0.00%	100.00%	100%	100.00%	100.00%	5.00%	0.00%
s420	36.70%	61.40%	38.9%	56.10%	70.33%	20.52%	59.50%
s444	0.33%	99.96%	99.6%	99.67%	99.74%	6.26%	2.33%
s510	1.56%	100.00%	98.4%	100.00%	100.00%	15.17%	32.36%
s526	0.00%	100.00%	100%	100.00%	100.00%	5.31%	0.25%
s5378	7.29%	97.74%	90.6%	93.95%	97.98%	14.26%	28.76%
s641	4.79%	98.61%	93.9%	96.22%	97.88%	13.29%	25.59%
s713	2.20%	97.21%	95.1%	97.80%	98.90%	11.39%	18.94%
s832	26.55%	71.50%	52.5%	68.95%	80.08%	17.96%	47.98%
s838	0.27%	94.79%	94.5%	98.90%	100.00%	8.24%	10.83%
s9234	0.92%	95.63%	94.8%	98.65%	99.68%	6.90%	5.28%
s953	7.01%	96.78%	90.0%	93.81%	96.67%	16.44%	38.50%
OR1200	15.5%	78.89%	66.7%	81.33%	90.46%	16.94%	44.20%
average	**12.59%**	**94.10%**	**83.22%**	**90.44%**	**95.19%**	**15.92%**	**39.65%**

As can be seen in Tab. 4.3, some of the circuits which are the small ones do not have an adequate number of correct estimations. This is mostly due to this fact that they are small circuits and they do not converge well with the given test vector limits and in many cases, their estimated range is too wide due to the small number of data points.

The results of 99,000 estimations have been also categorized based on each configuration of a circuit. Based on this categorization, shown in Tab.

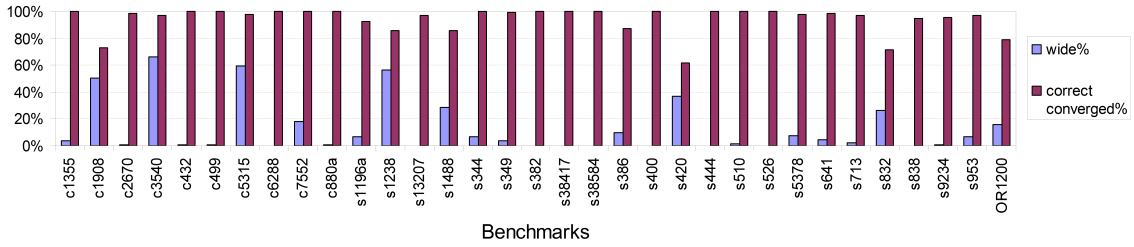


Figure 4.4: The percentage of wide vs. correct (and converged) estimations of EAGLE for each circuit

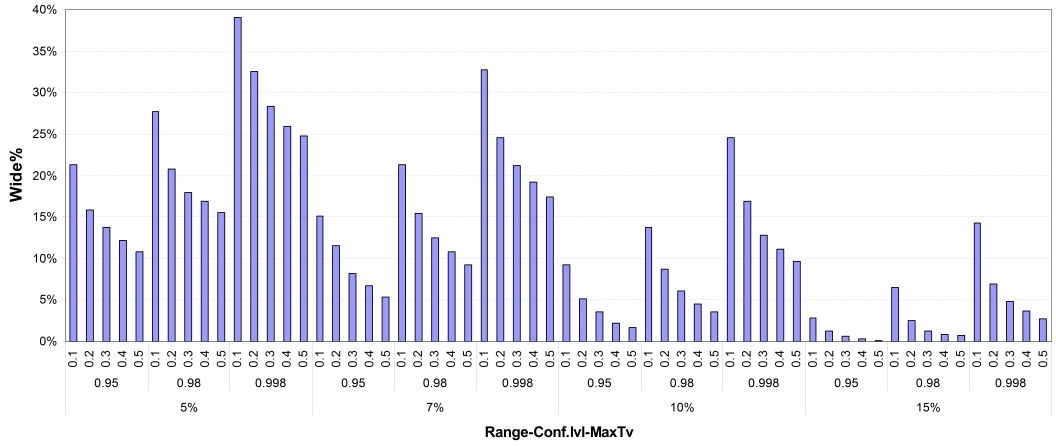


Figure 4.5: Wide range estimation rate based on each configuration

4.4, the average outcomes of all the circuits for each specified configuration is listed. This way, the effect of each constraint on the quality of estimation can be observed. As can be seen in Tab. 4.4 and Fig. 4.5, the number of wide range estimations decreases when the maximum test vector limit is increased. Also, the percentage of the correct estimations over converged cases or all cases increases with increasing the value for the maximum test vector limit (see Fig. 4.6 and Fig. 4.8). As can be seen in Fig. 4.6, when we increase the confidence level, the number of correct estimations increase (e.g., from 91% to 96% in case of desired range of 5% and maximum 10% of test vectors).

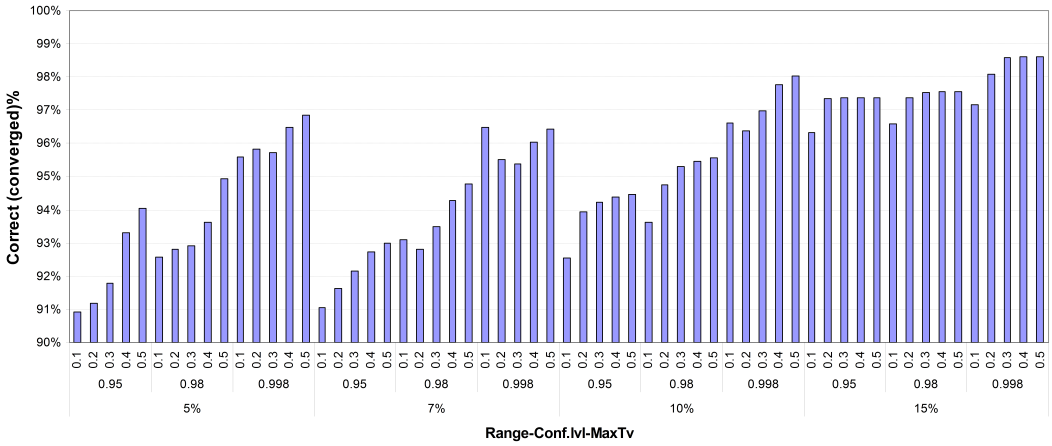
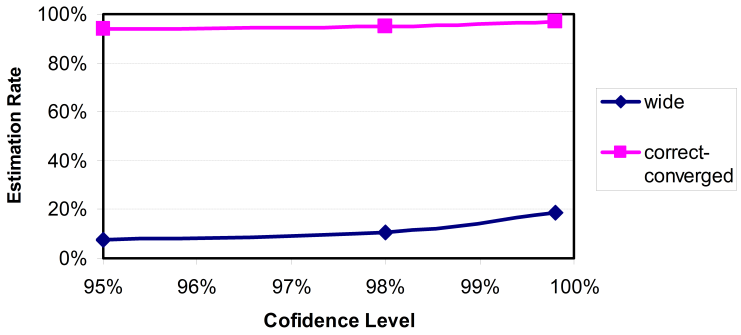


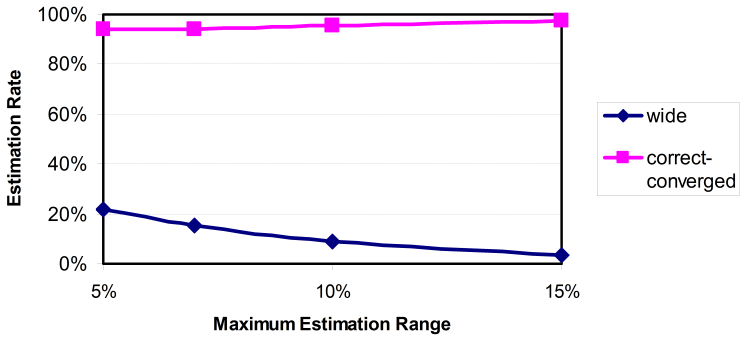
Figure 4.6: Correct (and converged) estimation rate based on each configuration

On average, among all 99,000 estimations, 7.6% of estimations have a wide range and 95% of the real coverage numbers lie in the estimated ranges that are not wide. 90% of all the estimations are correct or have 3% difference with upper or lower bounds, while 95% of all the estimations are correct or have 5% difference with upper or lower bounds of the estimated coverage range. As can be seen from our results, in almost 68% of cases, the algorithm converges before the defined test vector limit is reached. To get a better idea of how these outcomes change for each configuration, summarized figures, in which the correct and wide-range rates are averaged over two of the constraints are also demonstrated. This way gives a better idea of how each constraint changes these two rates (Fig. 4.7).

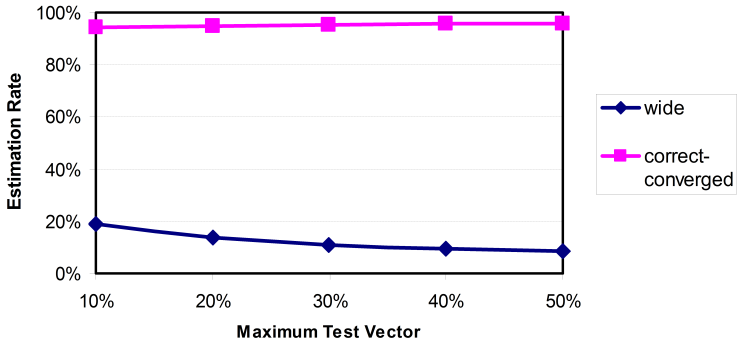
Figure 4.9 shows the estimated coverage range (using error bars) and real fault coverage numbers (shown with bars) for OR1200 processor for different random test vector sets for one configuration (confidence level=95%, maximum test vector limit=10% of all vectors, and desired range=5%). As can be seen in this figure, in most cases, the real fault coverage is within the



(a)



(b)



(c)

Figure 4.7: Averaged estimation results based on (a) confidence level, (b) range, and (c) maximum test vector constraints

Table 4.4: EAGLE estimation outcomes based on each configuration (averaged over circuits)

range	conf. level	max. tv%	wide%	$correct_{converged}$%	$correct_{all}$%	diff3%	diff5%	tv%	tv-lim%
5%	95%	10%	21.28%	90.91%	71.6%	86.26%	92.78%	8.38%	63.96%
		20%	15.85%	91.19%	76.7%	89.14%	95.97%	14.23%	54.89%
		30%	13.74%	91.78%	79.2%	90.86%	96.87%	19.46%	50.16%
		40%	12.20%	93.30%	81.9%	91.37%	97.19%	24.28%	46.96%
		50%	10.80%	94.05%	83.9%	92.01%	97.44%	28.87%	43.96%
	98%	10%	27.67%	92.58%	67.0%	84.22%	92.59%	8.67%	71.25%
		20%	20.83%	92.82%	73.5%	86.65%	94.82%	15.28%	62.49%
		30%	17.96%	92.91%	76.2%	88.18%	95.97%	21.21%	56.81%
		40%	16.93%	93.62%	77.8%	88.37%	95.85%	26.70%	53.55%
		50%	15.53%	94.93%	80.2%	88.95%	96.10%	31.97%	50.86%
	99.8%	10%	39.04%	95.60%	58.3%	76.55%	88.56%	9.09%	80.13%
		20%	32.59%	95.83%	64.6%	79.11%	90.54%	16.72%	73.61%
		30%	28.37%	95.72%	68.6%	81.28%	91.82%	23.84%	69.27%
		40%	25.88%	96.47%	71.5%	82.94%	92.40%	30.57%	65.69%
		50%	24.79%	96.86%	72.8%	82.88%	92.46%	37.08%	63.07%
7%	95%	10%	15.08%	91.05%	77.3%	87.41%	92.72%	7.74%	50.03%
		20%	11.57%	91.62%	81.0%	90.48%	95.65%	12.15%	39.94%
		30%	8.18%	92.14%	84.6%	91.88%	96.17%	15.87%	35.27%
		40%	6.77%	92.73%	86.5%	92.59%	96.55%	19.19%	32.01%
		50%	5.37%	92.98%	88.0%	92.91%	96.55%	22.36%	30.48%
	98%	10%	21.28%	93.10%	73.3%	85.62%	92.52%	8.12%	58.59%
		20%	15.40%	92.82%	78.5%	88.75%	95.08%	13.33%	48.43%
		30%	12.46%	93.50%	81.8%	90.35%	96.23%	17.89%	43.32%
		40%	10.80%	94.27%	84.1%	91.12%	95.97%	22.03%	39.81%
		50%	9.27%	94.79%	86.0%	91.95%	96.23%	25.89%	36.74%
	99.8%	10%	32.72%	96.49%	64.9%	79.36%	88.95%	8.64%	70.54%
		20%	24.54%	95.51%	72.1%	82.94%	91.31%	15.11%	60.96%
		30%	21.21%	95.38%	75.1%	84.98%	92.59%	20.88%	55.27%
		40%	19.23%	96.04%	77.6%	85.69%	92.91%	26.19%	51.31%
		50%	17.44%	96.44%	79.6%	86.26%	93.04%	31.26%	48.88%
10%	95%	10%	9.27%	92.54%	84%	90.80%	94.06%	6.93%	33.16%
		20%	5.18%	93.94%	89.1%	93.61%	96.42%	9.67%	24.60%
		30%	3.58%	94.23%	90.9%	94.12%	96.36%	11.95%	21.66%
		40%	2.24%	94.38%	92.3%	94.57%	96.74%	13.99%	19.11%
		50%	1.73%	94.47%	92.8%	94.70%	96.68%	15.79%	16.42%
	98%	10%	13.74%	93.63%	80.8%	88.69%	93.23%	7.33%	41.85%
		20%	8.75%	94.75%	86.5%	91.63%	95.72%	10.91%	32.14%
		30%	6.07%	95.31%	89.5%	92.91%	96.29%	13.85%	27.99%
		40%	4.47%	95.45%	91.2%	93.55%	96.23%	16.52%	25.69%
		50%	3.58%	95.56%	92.1%	93.93%	96.49%	19.05%	24.22%
	99.8%	10%	24.60%	96.61%	72.8%	83.07%	89.97%	8.05%	56.04%
		20%	16.87%	96.39%	80.1%	87.48%	92.84%	13.08%	46.71%
		30%	12.84%	96.99%	84.5%	89.71%	93.87%	17.42%	41.21%
		40%	11.18%	97.77%	86.8%	90.67%	94.38%	21.34%	37.12%
		50%	9.65%	98.02%	88.6%	91.25%	94.31%	24.96%	34.70%
15%	95%	10%	2.88%	96.32%	93.5%	96.74%	97.89%	6.16%	18.59%
		20%	1.21%	97.35%	96.2%	97.96%	98.79%	7.50%	10.10%
		30%	0.58%	97.37%	96.8%	98.15%	98.98%	8.32%	6.45%
		40%	0.32%	97.37%	97.1%	98.15%	98.98%	8.86%	4.60%
		50%	0.13%	97.38%	97.3%	98.34%	99.04%	9.28%	3.51%
	98%	10%	6.52%	96.58%	90.3%	95.08%	97.12%	6.48%	24.35%
		20%	2.56%	97.38%	94.9%	97.00%	98.21%	8.48%	17.64%
		30%	1.28%	97.54%	96.3%	97.57%	98.66%	10.00%	12.78%
		40%	0.83%	97.55%	96.7%	97.96%	98.66%	11.13%	10.10%
		50%	0.70%	97.55%	96.9%	98.02%	98.72%	12.05%	8.12%
	99.8%	10%	14.25%	97.17%	83.3%	89.58%	93.99%	7.18%	38.21%
		20%	6.96%	98.08%	91.3%	94.12%	96.36%	10.34%	27.54%
		30%	4.79%	98.59%	93.9%	95.14%	96.81%	12.88%	23.90%
		40%	3.64%	98.61%	95%	95.85%	97.19%	15.17%	21.85%
		50%	2.68%	98.62%	96%	96.61%	97.32%	17.28%	19.87%

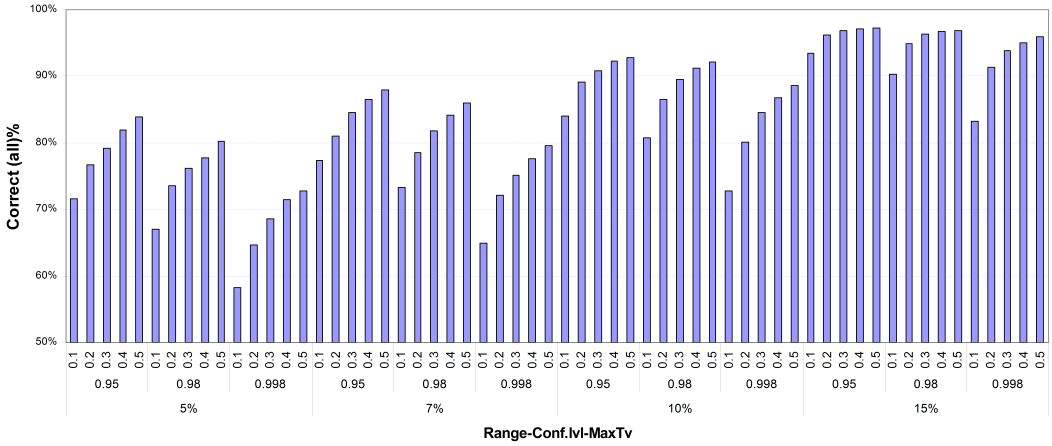


Figure 4.8: Total correct estimation rate based on each configuration

estimated range by EAGLE.

As can be seen in the tables and figures in this section, using this regression model can be very promising due to the fault simulation speedup. For example, for the case of maximum test vector limit equal to 10% of the test vectors, the average test vector percentage used in partial fault simulation is 7.6%. In this case, 94% of estimations exactly lie in the estimated rate. The run times of partial and full fault simulation (shown as *partial-FS* and *full-FS* columns) have been shown for 20 different test vector sets for OR1200 in Tab. 4.5. This table is based on the configuration of maximum test vector equal to 10% of the test vectors, confidence level equal to 95%, and required range equal to 5%. As can be seen in this table, on average, the ratio between full fault simulation and partial fault simulation run times is more than one order of magnitude.

It should be noted that the overhead for GIC calculation and one good simulation pass is not considered above. One pass of good simulation (with

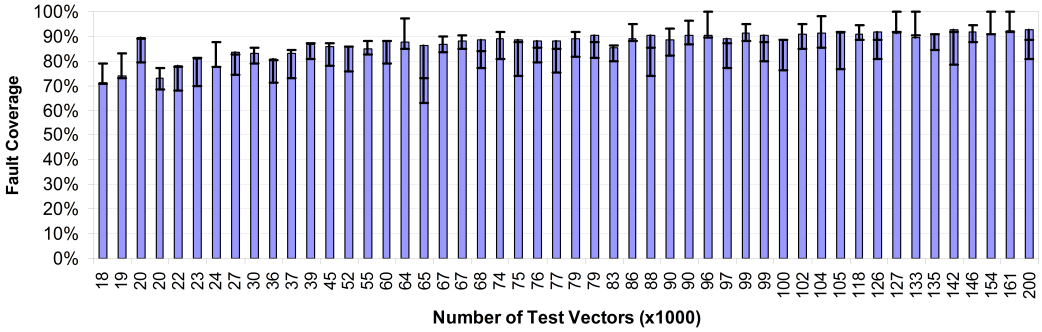


Figure 4.9: EAGLE estimation of OR1200 for a configuration in which range=5%, confidence level=95%, and max. test vector=10%. The bars represent real fault coverage while the error bars show the estimated coverage range

Table 4.5: EAGLE fault simulation run time vs. full fault simulation run time for OR1200 for a configuration in which range=5%, confidence level=95%, and max. test vector=10%

"Number of" "test vectors"	partial-FS (seconds)	full-FS (seconds)	ratio (full-FS/partial-FS)
104870	1611	14638	9.08
105930	1355	12907	9.52
126860	1954	18088	9.25
135230	1297	11655	8.98
18660	322	6104	18.95
20900	688	6416	9.32
22150	370	8727	23.58
23300	507	6122	12.07
27650	822	7794	9.48
30000	746	7122	9.54
36840	970	8691	8.95
45840	757	6528	8.62
52010	1490	13192	8.85
64030	1827	13857	7.58
67160	600	5986	9.97
68240	814	7972	9.79
75150	1010	9261	9.16
79380	1094	9610	8.78
Geo.Mean			**10.16**

no injected faults) is relatively very fast so that it can be easily ignored. For example, for OR1200 processor with 83,000 test vectors, the time consumed for fault simulation is around 14,000 seconds, while one pass of good simulation takes 47 seconds to complete. So far, GIC coverage is extracted from the VCD (Value Change Dump) waveform file. This method is quite slow compared with accessing internal data structures using VPI (Verilog Procedural Interface). In the example mentioned above, GIC calculation takes around 700 seconds. However, with an efficient VPI-based implementation, GIC calculation time should be comparable to the time for one pass of simulation. It should be also mentioned that the overhead of regression calculation is very low. For example, calculating the regression model and coverage estimation for all 99,000 cases takes around 5 hours for the above experiments. Therefore, on average, each calculation takes a fraction of a second to complete. In fact, for small circuits, this calculation takes less than 0.1 seconds, while it takes a few seconds for larger circuits with a larger number of test vectors (such as the case for OR1200 circuit for 50% of its entire test vector set).

4.5 Conclusions

This chapter, introduced EAGLE, which is a linear regression-based fault coverage estimation model. For a circuit under test and a set of test vectors, EAGLE uses fault simulation for a portion of the test vectors and one pass of fault-free simulation and calculates the best fit line between the partial fault coverage and a statistical metric called GIC, which counts the number of new gate input combinations for each test vector. Using statistical formulae, EAGLE finds a lower and an upper bound for the estimated fault coverage based on a few constraints that can be set by the user. The experiments in

this chapter show that EAGLE is a promising method for estimating fault coverage. For example, by fault simulating about 7.6% of the test vectors, the real fault coverage values lie between the estimated bounds in about 94% of the cases.

Chapter 5

Soft Error Vulnerability Analysis by Local Simulations

5.1 Introduction

In this chapter, a novel soft error vulnerability estimation called RAVEN (RApid Vulnerability EstimatioN) is introduced [56]. RAVEN takes advantage of statistics from fast local simulations[1] to build a toolflow that can calculate the detection probabilities of soft error candidates in the whole system. In RAVEN, the probability of DUE (Detected Unrecoverable Error) and SDC (Silent Data Corruption) outcomes for all possible soft error candidates in a period of time is calculated. In this chapter, single bit-flip on a flip-flop is used as the error model since the result of error injection with this error model is close to radiation-based injections [11, 78]. On the other hand, the results of high-level error injection are considerably different from the results of the injection for this error model [19, 55]. Therefore, I avoid using such high-level error models in this chapter.

In this chapter, RAVEN run times and outcome probabilities are compared with error injection for a sample of error candidates. Different sample sizes are used in this analysis, and for large enough sample sizes (required for

[1]These local simulation passes are used in a same way as FALCON methodology. The essence of soft errors makes the usage of local simulations even more efficient compared to manufacturing fault coverage estimation.

an acceptable margin of error), RAVEN runs 2 to 3 orders of magnitude faster than error injection. In the experiments in this chapter done on the IVM processor model [92] with SPECINT2000 workloads, all of the benchmarks lie in the outcome range that is calculated by error injection, except for the workload *crafty*. To investigate further, I have performed complete error injection targeting a small subset of error candidates and 400 cycles in *crafty* and compared the results of flip-flop vulnerability factors[2] that RAVEN calculates to those in complete injection and sampling error injection. These results show the accuracy and speed of RAVEN over sampling error injection when detailed vulnerability is necessary for deciding on resilience techniques for a design.

RAVEN can be used with existing techniques to further improve the efficiency of soft error analysis. For example, RAVEN can be used in FPGA-based systems or hierarchical simulation environments. To show the speed advantage of RAVEN in a proven, efficient simulation environment, we use a hierarchical simulation environment as described in [92]. This environment can accelerate error propagation by switching between an instruction-set simulator and Verilog RTL simulator. All of our simulations (RAVEN and error injection) are done in this hierarchical simulation environment.

In Section 5.2, RAVEN methodology is discussed. Section 5.3 describes the related experimental results. Section 5.4 shows the speed-up of RAVEN over complete error injection and the effectiveness of RAVEN in analyzing designs for resilience.

[2] $\frac{\#\ of\ erroneous\ outcomes}{\#\ of\ total\ injections}$ for errors injected into a flip-flop

5.2 RAVEN Methodology

As mentioned above, RAVEN is a statistical method to estimate the probability of DUE and SDC outcomes for all possible soft error candidates in a period of time. RAVEN works faster than traditional error injection methods. However, using local fault simulations and probabilistic calculations, this method could introduce inaccuracies. An apples to apples comparison of RAVEN with existing error injection techniques would require performing error injection for every possible soft error candidate for long periods of execution. Since this would require inordinate amounts of time, a complete error injection has been done for a short period and a small sub-set of flip-flops. Comparing the results of complete error injection and RAVEN shows that RAVEN can make more accurate estimates of the flip-flop vulnerability factors (and also average outcomes) than error injection with sampling.

5.2.1 RAVEN Steps

RAVEN takes advantage of local simulations which are significantly faster than simulating the entire system. By following the four steps listed below, the outcomes for given workloads under soft error candidates can be estimated in a period of time. The design is first partitioned, using the design hierarchy as a guide, so that each partition contains one or more modules/entities of the design, depending on their size. For example, in a pipeline-based processor, each pipeline stage can be one partition. Partitioning a design for RAVEN is not limited to the pre-defined modules; any partitioning algorithm can be used in RAVEN. However, the more system-level feedback loops and system-level reconvergent fanouts are avoided, RAVEN estimation will be more accurate. Also, large partitions is better to be avoided since they

will degrade RAVEN run-time efficiency. Below are the steps performed in RAVEN.

Step 1: Performing one pass of *golden* simulation (with no error injection) for the whole system and storing the values for each partition's inputs/outputs in a file.

Step 2: Building a table (**propagation table**) for each partition P which shows the probability of error propagation from each input of P to each output of P, based on the values from the previous step.

Step 3: Building a table (**detection table**) for each partition P which shows the probability of error propagation from each source of error inside P to each of P's outputs, based on the values in Step 1.

Step 4: Calculating the detection probability of each soft error candidate in the entire system.

The following sections will describe the above steps in more detail.

5.2.2 Propagation Tables

Propagation tables show the error propagation probability from each input to each output of a partition. According to this definition, the number of times an error at one of the inputs is observed at any outputs should be calculated. For this purpose, single stuck-at faults are injected one at a time at each input of a partition and the number of times this error is observed at any output is calculated. Then, the probabilities of error propagation are defined by dividing the number of error detections at each output, by the number of cycles. This number is named the propagation probability of an error on a local input to a local output. For example, in Fig. 5.1, partition P has n inputs and

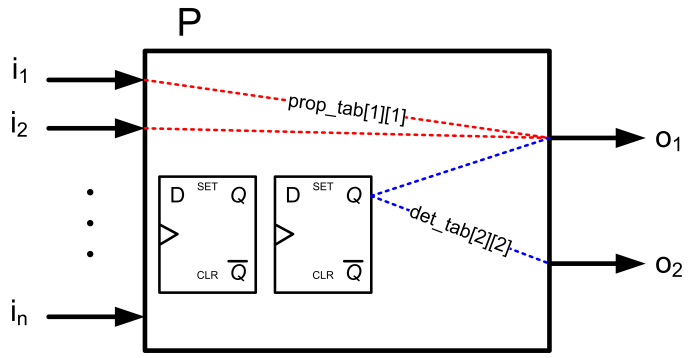


Figure 5.1: A sample module with n inputs, 2 outputs, and 2 flip-flops.

2 outputs and suppose P is being analyzed for 10 clock cycles. To generate P's propagation table, $2 \times n$ simulation passes are performed and a stuck-at 1 and a 0 on each input, for each simulation pass, are injected. In each simulation pass, the number of times that the values of o_1 or o_2 are not the same as their stored golden values are counted. Then the propagation probability from i_1 to o_1, for instance, is calculated by dividing the number of times that o_1 is different than its golden value when i_1 is faulty, by 10 (the number of cycles). Equation 5.1 shows the propagation probability from input i to output o of a partition. In this equation, *i_sa_0/1* means line i stuck-at-0 or stuck-at-1. We add these two cases to obtain a rough estimate of the probability of an error on the output when there is an error on each input [3].

$$prop\_table[i][o] = \frac{\#\ of\ cycles\ i\_sa\_0/1\ detected\ on\ o}{total\ \#\ of\ cycles} \tag{5.1}$$

[3]If the direction of the bit-flip ($1 \rightarrow 0$ or $0 \rightarrow 1$) is also assume in probability tables, it will result in a more precise estimation. However, this needs 4x more storage than the current rough estimation.

It should be noted that a fault simulation tool (instead of injecting the errors into inputs serially) can be used. This way, one pass of fault simulation is enough to calculate this table.

5.2.3 Detection Tables

A detection table for each partition P indicates the probability that a soft error in P will be seen at one or more outputs of P. Similar to propagation table elements, each element of detection table can be calculated by injecting a stuck-at fault into a flip-flop, simulating the partition in presence of that fault, and comparing each output with its golden value at each cycle. Dividing the number of observations of an error on an output by the total number of cycles will give the probability of detection of that error for that output. However, this number shows the probability of an error detection when that error is present on the flip-flop in all cycles. To distinguish soft errors from stuck-at faults, it is usually assumed that a soft error affects the output of a flip-flop in just one (or a few) cycle(s). Therefore, the original detection probability should be multiplied by $\frac{error\ duration}{\#\ of\ cycles\ error\ can\ be\ present}$ to model a soft error. It is assumed that the error duration is 1 cycle in this chapter. Such an assumption is not necessary in propagation tables, since RAVEN is calculating the average propagation of an error from a local input to a local output. One advantage of RAVEN over error injection is that if there is a need to analyze design vulnerability for soft errors with different error durations, RAVEN only needs to re-calculate the detection probabilities without needing to re-run the local simulations. In Fig. 5.1, to generate the detection table of partition P, 4 passes of simulation for flip-flop stuck-at-0 and stuck-at-1 faults should be done. Then the number of erroneous outputs observed for each pass is counted and divided

by 10 (number of cycles). To consider these errors as soft errors this number is multiplied by $\frac{1}{10}$. Equation 5.2 shows the detection probability for error e to output o. o is an output of a partition and e is a soft error candidate in that partition. Usually, the number of cycles that e can happen is equal to the total number of cycles.

$$det\_table[e][o] = \frac{\#\ of\ cycles\ e\_sa\_0/1\ detected\ on\ o}{total\ \#\ of\ cycles} \times \frac{duration\ of\ e}{\#\ of\ cycles\ e\ can\ happen} \tag{5.2}$$

NOTE: Using stuck-at faults to model soft errors can result in larger vulnerability factors in general, since there are feedback paths in each module and the error propagation for a stuck-at fault can be potentially more than a soft error. Therefore, the calculated vulnerability factors by RAVEN can be considered as an upper bound for vulnerability factors.

5.2.4 System Level Detection Probability Calculation

After generating the propagation and detection tables for each partition, the detection probability for each soft error on a flip-flop in the whole system should be calculated by using the detection and propagation tables of each partition. First, to calculate the outcome of an error e at the system level, RAVEN picks the detection table elements in partition P corresponding to e. Then it updates the output values of partition P due to these elements. Next, RAVEN updates the outputs of each partition that have inputs connected to the outputs of P by using their propagation probability tables. This calculation is continued until RAVEN reaches a signal of interest, which will be discussed in Section 5.2.5. The probability observed on this signal is the

detection probability of e in the whole system. This process is performed for each flip-flop of interest.

To calculate the detection probability of the outputs of a partition due to its input detection probabilities, RAVEN uses a simple probability union calculation based on the input probabilities and the corresponding propagation table elements. This calculation is shown in Eq. 5.3. RAVEN uses this equation to calculate the detection probability of an error on output o in partition P.

$$det\_prob[o] = 1 - \prod_{i=1}^{I_p}(1 - prop\_tab_p[i][o] \times det\_prob[i]) \tag{5.3}$$

In Eq. 5.3, I_p is the number of inputs of partition P, $prop\_tab_p[i][o]$ is the propagation probability from input i to output o, and $det\_prob[i]$ is the observed detection probability on input i.

5.2.5 Outcome Probability Calculation

In soft error injection, we observe the outcome of a specified workload with an error candidate injected in the design. The outcomes discussed in this chapter are SDC and DUE outcomes. In the former, the program exits normally, but the output of the program is not correct, or the state of the system is not equal to the golden state. In the experiences in this chapter, if an error stays in the system for a long time it is very likely that it causes an SDC outcome (as seen in more than 93% in SPEC workloads running on IVM processor model). Therefore, in order to speed up the evaluation, the state of the system after a certain amount of cycles is compared with the golden state, and if they do not match, a dummy output called "sdc" becomes '1'.

In the latter case (DUE), the program exits prematurely. RAVEN checks all the conditions which cause a program to halt (e.g., instruction exception) and asserts a dummy output, called "halt", under any of these conditions. Since RAVEN performs local simulations, it is not able to find out some system-level cases that make the program to run infinitely (i.e., cause a "hang" outcome, for example). Although this outcome happens very rarely, it should be considered as a source of inaccuracy in RAVEN.

In IVM processor model, which is used in experiments in this chapter, if an error causes the pipeline to stall persistently or causes an exception in a clock cycle, this exception will happen at every clock cycle from that cycle until the end of the simulation. Therefore, the propagation probabilities in these cases are skewed to the cycles closer to the end of simulation. In calculating the probability of a "halt" outcome based on the probability of its corresponding partition's inputs, RAVEN simply calculates the maximum propagation probability from each input path to the "halt" output instead of calculating the "halt" output probability based on Eq. 5.3. As a future work, the skew can be consider in every calculation and, therefore, Eq. 5.3 needs to be altered to consider the skew as well as the probability values.

When RAVEN calculates the "halt" and "sdc" detection probabilities for each flip-flop, it calculates the outcome probability for DUE and SDC outcomes for a set of error candidates. From the definitions of detection and propagation probabilities, each outcome detection probability for each flip-flop gives a rough estimate of how many times a soft error on a specified flip-flop causes an erroneous outcome during simulation. Therefore, if the detection probabilities on the "sdc" and "halt" dummy outputs are averaged over the number of flip-flops, we can have outcome probabilities (an estimation

of outcome rate) for all soft error candidates (like Eq. 5.4 for SDC outcome).

$$sdc\_outcome = \frac{\sum_{f=1}^{FF\_num} det\_prob_f^{sdc\_dummy\_out}}{FF\_num} \tag{5.4}$$

An advantage of RAVEN is that the detection probability of each error in the whole system can represent the vulnerability factor for its respective flip-flop as well. These vulnerability factors, calculated for the flip-flops of a design, can be used as a guide to choose an efficient resilience technique for that design.

5.3 Experimental Results

I have implemented RAVEN and applied it to IVM [92], which is an out-of-order and super-scalar Alpha-based processor. In order to partition this processor, each stage of the pipeline is considered as a partition, and the rest of the "glue" logic is put into a separate partition. The top-level Verilog model, which loads a program into IVM memory and starts the processor, has been changed so that it can store the golden input and output values of each partition in a file, as well as the golden state of the processor on the cycle that local simulations are started. These files are generated during one pass of golden simulation (simulation with no error injection). These files are used in the table generation step. For propagation table generation, I have developed a Verilog wrapper for each partition. This wrapper reads the golden inputs/outputs from the files stored in the golden simulation step, changes one input and simulates the Verilog model of that partition. Then at each cycle, it compares the output with the golden output, and for the outputs that are different from the golden output, the corresponding table

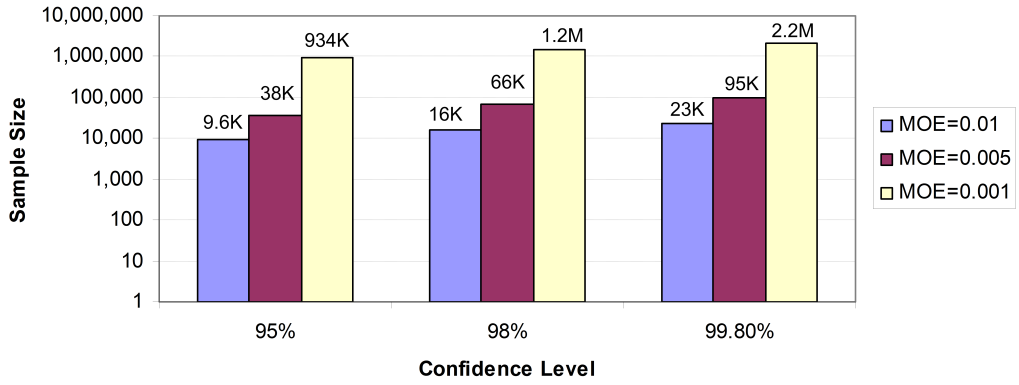


Figure 5.2: Maximum sample sizes for different absolute MOE values.

element is updated. After one simulation pass, the system is reset to its initial state and the model is ready for the next input error injection. Detection table generation is similar to propagation table generation; however, instead of input error injection, the error is injected on each flip-flop in that partition. Since the RTL (Register Transfer Level) of IVM was available in this research, I have done table generation step by performing one RTL simulation pass for each error injection. If the synthesizable model was available, a fault simulator could be used to generate the table for each partition.

For system-level probabilistic calculation, I have developed a SystemVerilog model which simply replaces each *wire/reg* signal type with *real* type so that they can pass probability values instead of logic values. Each partition is replaced by a process, sensitive to all of its inputs. Inside this process is a function implementing Eq. 5.3. Also, all propagation tables and detection tables are included in this model.

In this chapter, RAVEN has estimated the outcome probabilities for

some parts of SPECINT2000 workloads, with MinneSpec input data, on IVM. In this section, RAVEN results are compared with statistical fault injection (SFI [40]) results. SFI uses samples of error candidates in the injection process. In this experience, SFI follows the mixed-level methodology used in [92], which runs the program in RTL for a limited number of cycles and the rest of the program runs in an instruction set simulator. Error injections are done only during RTL simulation. Therefore, RAVEN also runs for this limited simulation period[4]. Figure 5.3 shows the steps followed for each injection in SFI. As mentioned, these steps are the same as the steps followed in [92].

RAVEN has been executed on a Dell™ PowerEdge R720 system, with two Intel™ Xeon E5-2690 (2.90 GHz) and 384 GB of random access memory. On the other hand, all error injections for SFI have been executed on one of the Texas Advanced Computing Center (TACC) systems (Stampede supercomputer [3]). Therefore, to compare RAVEN and SFI run times, I needed to extrapolate the run times for SFI as the number of injections multiplied by the time spent for one good simulation on the Dell™ machine that RAVEN was executed. Note that in this extrapolation, I have not counted cases for program time-outs and the time for injecting an error. Therefore, this extrapolation is fair for SFI. Each benchmark has a different run time, since they have different number of instructions. All numbers for SFI run times follow the extrapolation mentioned above.

In this chapter, RAVEN is compared to SFI with different sample sizes. Due to [40], the sample size (n) can be calculated using Eq. 5.5. In this equa-

[4]In this chapter, the injections are done during 2400 cycles. However, the efficiency of RAVEN becomes even more apparent if the number of cycles in the RTL simulation are increased.

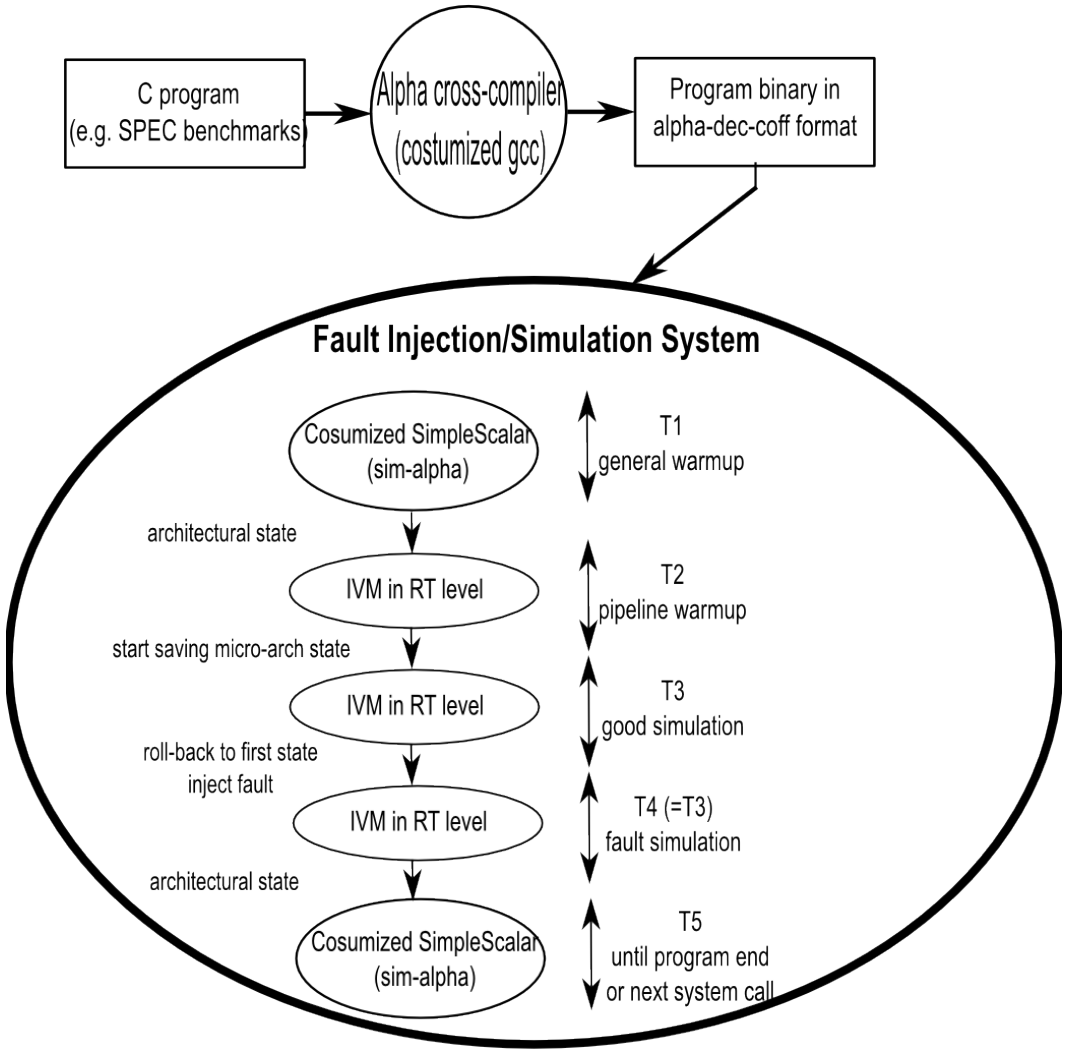


Figure 5.3: Error injection steps in IVM. T1 to T5 are the periods spent for each step. These time periods can be different for different programs running on IVM.

tion, e is the absolute margin of error (MOE) of sampling. p is the estimated outcome rate that we gain. This number is usually replaced by 0.5 to gain the maximum sample size, since it is not known initially what outcome rates are going to be gained in SFI. N is the set of all error candidates, and t is a cut-off point corresponding to a confidence level. In this chapter, my calculations are based on 95%, 98%, and 99.8% confidence levels (with corresponding t=1.96, 2.5758, and 3.0902).

Using Eq. 5.6, MOE can be calculated for a given sample size, a known outcome rate, and a given cut-off point.

$$n = \frac{N}{1 + e^2 \times \frac{N-1}{t^2 \times p \times (1-p)}} \tag{5.5}$$

$$e = t \times \sqrt{\frac{p \times (1-p) \times (N-n)}{n \times (N-1)}} \tag{5.6}$$

Using Eq. 5.5, sample sizes for 3 confidence levels and three different MOEs (0.01, 0.005, and 0.001) are calculated. This is shown in Fig. 5.2. As can be seen in this figure, the sample size grows significantly when decreasing the MOE. Due to relatively small outcome rates for DUE and SDC, which are typically between 2% to 14%, even a small MOE can cause a quite large percent relative MOE in outcome calculations. For example, if in an injection experience, SDC rate is equal to 4% and the absolute MOE for that sample size is 0.01 (=1%) it means that the SDC outcome in this case lies between 4%±1%. This means that the percent relative MOE in SDC rate calculation is 25%.

Based on the sample sizes in Fig. 5.2, SFI has been performed for three confidence levels and MOE equal to 0.01. The percent relative MOE of the

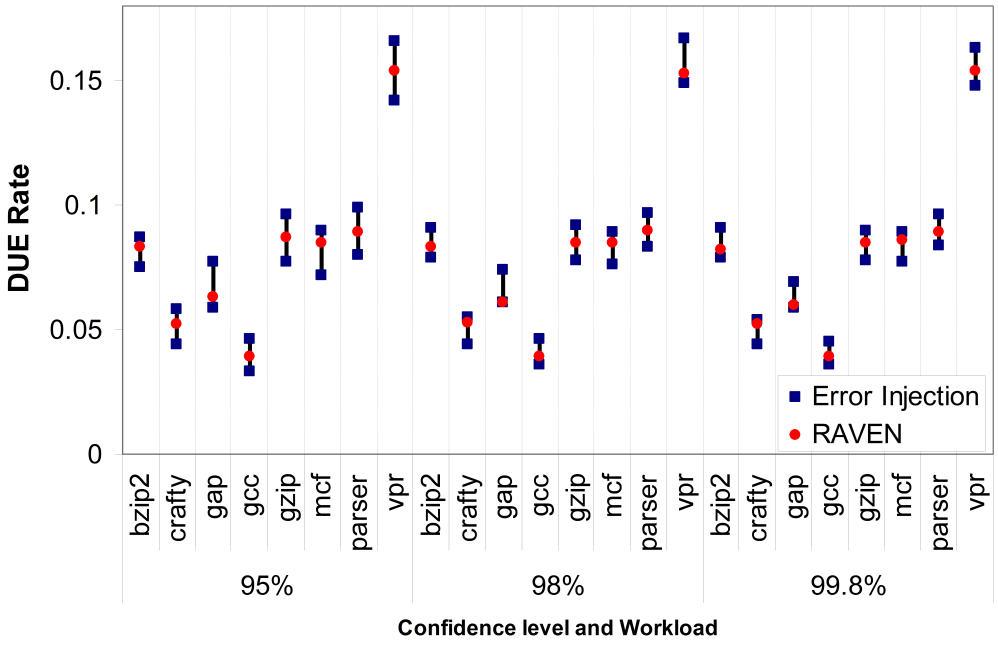


Figure 5.4: Range of DUE rates in SFI and DUE probability in RAVEN (MOE=0.01).

calculated SDC and DUE rates are between 5% to 22% in this experience. Therefore, to have small percent relative MOEs, we need to set e to even a smaller number.

DUE and SDC outcome probabilities are estimated for all the error candidates in a time interval for 8 SPECINT2000 workloads on IVM using RAVEN method and these outcome rates are calculated using SFI for 3 different sample sizes (for different confidence levels and MOE=0.01). In Figures 5.4 and 5.5 the outcome rates of RAVEN vs. SFI for these workloads and each sample size are shown. In Fig. 5.6, the run times of RAVEN and SFI are compared. RAVEN run times are different for each sample size, since the

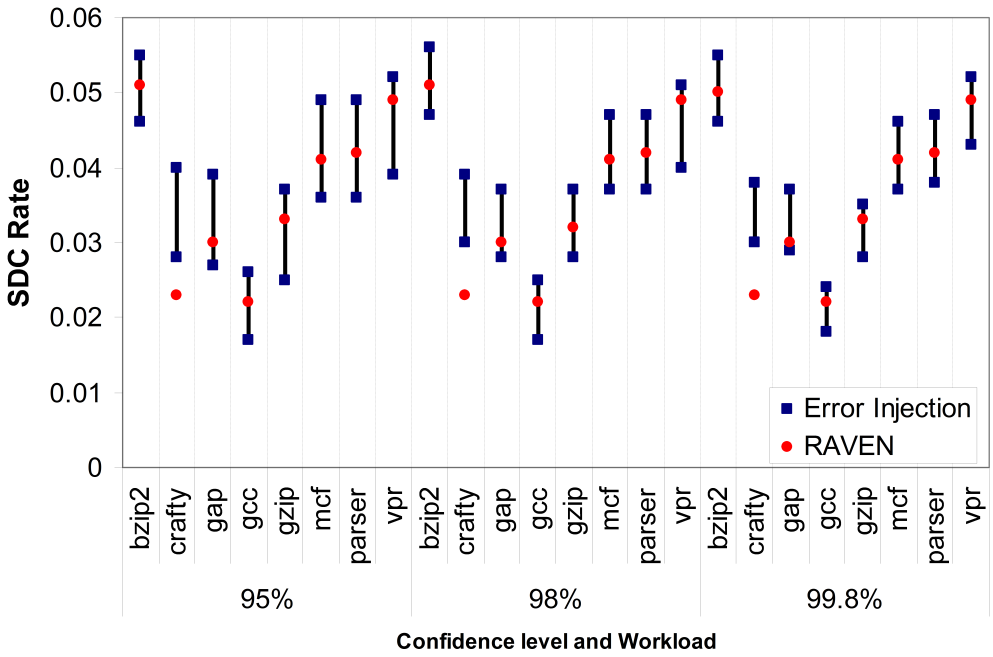


Figure 5.5: Range of SDC rates in SFI and SDC probability in RAVEN (MOE=0.01).

outcome probability for only those flip-flops that were used in SFI samples are considered. This way results in a more precise comparison between RAVEN and SFI. Even for the smallest sample, RAVEN works more than 17x faster than SFI, on average. Also, the SFI run times for larger samples (i.e., smaller MOE values) are extrapolated and compared to RAVEN run times (shown in Fig. 5.2). Speed-up (ratio of SFI run time to RAVEN run time) is calculated for each workload and the average speed-up values using, geometric mean formula, is shown in Fig. 5.7.

As shown in this section, RAVEN can estimate the outcomes of soft error candidates in a system mostly within the range indicated by SFI. Due to

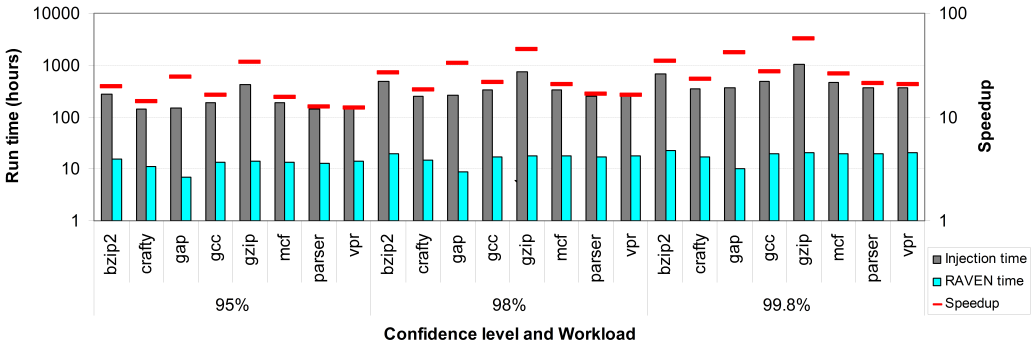


Figure 5.6: Run times for RAVEN and SFI methods, along with the speed-up (MOE=0.01).

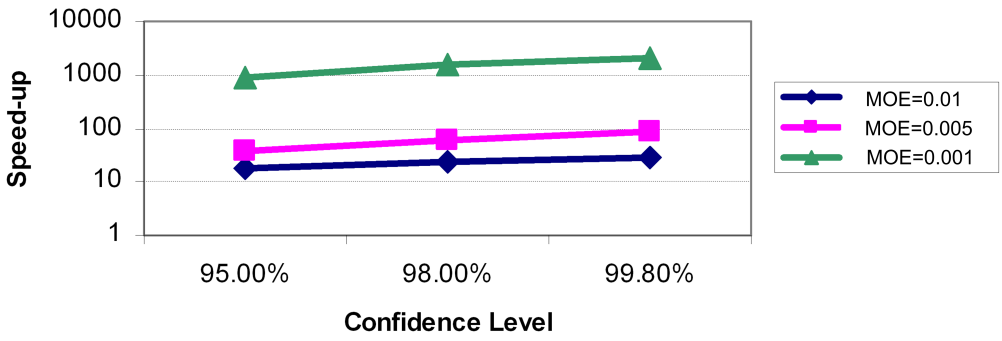


Figure 5.7: RAVEN speed-up, compared to SFI, for different confidence levels and MOE values.

Fig. 5.5, only SDC rate in crafty application is out of calculated range of SFI and it is still close to the minimum value of the calculated range of SFI. Results in Fig. 5.7 show that RAVEN can estimate the outcome rates 1 to 3 orders of magnitude faster than SFI, depending on the sample size. These calculations are based on the assumption that we use only one computation resource. To catch a glimpse of the efficiency of RAVEN, suppose we distribute RAVEN processes among **15** computing resources (because we have 15 partitions). Then, we would need on average, **253** computing resources for SFI in order to take the same time as RAVEN (for confidence level=95% and MOE=0.01).

5.4 Vulnerability Factor Analysis

5.4.1 Complete Error Injection vs. RAVEN

The ideal way for measuring the vulnerability of a system is to calculate the outcome rates for any possible soft error that can happen in a system, which we call *complete error injection*. To have a fair comparison, results for RAVEN should be compared with error injection. Therefore, we need to perform a complete error injection during the time period that RAVEN was running and compare its outcome rates and run time with RAVEN outcome probabilities and run time. Since there are 14,184 flip-flops in IVM that are used for error injection, more than 33,600,000 injections are needed for this fair comparison, which is obviously not possible to do in a reasonable time. Table 5.1 gives an idea of complete injection run times. This table compares the extrapolated complete error injection and RAVEN run time for each workload and it also includes the geometric mean of all workloads in the last row. Table 5.1 shows over 32,000x speed-up over the complete error injection. Obviously, it is not possible to compare the outcome rates of complete error

Table 5.1: Run time comparison for RAVEN vs. complete error injection.

Application	RAVEN run time (hrs)	Complete injection run time(hrs)	speed-up
bzip2	29.16	1100678	37741
crafty	22.48	566319	25186
gap	13.26	601117	45333
gcc	25.88	762815	29474
gzip	26.97	1645344	61006
mcf	26.30	744092	28297
parser	25.24	576437	22835
vpr	26.90	594782	22107
G. Mean	**23.95**	**767161**	**32026**

injection with RAVEN's outcome probabilities, unless we perform a complete error injection. But in the next section (Section 5.4.3), the results for outcome rates of a small sub-set of flip-flops for complete error injection and RAVEN are shown and they are compared with the results for SFI.

5.4.2 SFI and Flip-flop Vulnerability Factors

As discussed in Section 5.3, in SFI there is a need to find the appropriate sample size based on the goal of error injection (which can be average outcome calculation for all flip-flops or calculating the most vulnerable flip-flops for adding partial resilience to the design). This section shows the amount of error (MOE) introduced by SFI in flip-flop vulnerability analysis (vs. average vulnerability) if the same sample size used for average outcome calculation is used.

Soft error candidates are distributed in two dimensions: flip-flops (spatial) and clock cycles (temporal). If we want to calculate the average outcome of error injection while a set of benchmarks are running on the design, we can

take a sample of candidates distributed in both spatial and temporal dimensions. If the number of flip-flops in a design is d, the total number of necessary clock cycles for running all the applications is c, which makes the population size equal to $d \times c$, and the sample size is n, due to [40], sampling MOE can be calculated as in Eq. 5.7.

$$e = t \times \sqrt{\frac{p \times (1-p)}{n} \times \frac{c \times d - n}{c \times d - 1}} \tag{5.7}$$

On the other hand, if we use the sample with size n for finding the most vulnerable flip-flops in the design, it is likely that each flip-flop is error injected around $\frac{n}{d}$ times, on average. Therefore, if we want to calculate the MOE for the injection, based on each flip-flop individually, the sample size is $\frac{n}{d}$, and the population size is c. Therefore,

$$e' = t' \times \sqrt{\frac{p \times (1-p)}{\frac{n}{d}} \times \frac{c - \frac{n}{d}}{c - 1}} \tag{5.8}$$

If the confidence level in both e in Eq. 5.6 and e' are considered the same, i.e., $t = t'$, then by combining Equations 5.6 and 5.8, e' becomes as a function of e, the number of flip-flops in the design, and the number of clock cycles needed for running the applications.

$$e' = e \times \sqrt{\frac{c \times d - 1}{c - 1}} \tag{5.9}$$

As an example,

d = 3000

c = 200,000

e = 1%

confidence level = 95%

Then, e' with 95% confidence will be:

$$e' = 0.01 \times \sqrt{\frac{200000 \times 3000 - 1}{199999}} \approx 0.55 \tag{5.10}$$

Equation 5.10 shows that an absolute MOE of 1% in calculating average outcomes results in an absolute error of 55% (on average) if the vulnerability of each flip-flop is calculated with the same sample used for calculating the average outcomes. This error will be even higher if there are more flip-flops in the design. In this example, if each flip-flop vulnerability is required to have 1% of absolute MOE with 95% confidence level, n' samples for each flip-flop is needed. n' can be calculated as in Eq. 5.11 (similar to 5.5).

$$n' = \frac{c}{1 + \hat{e}^2 \times \frac{c-1}{t^2 \times p \times (1-p)}} = \frac{200000}{1 + 0.01^2 \times \frac{199999}{1.96^2 \times 0.5 \times (1-0.5)}} \approx 9164 \tag{5.11}$$

Therefore, a total number of $9164 \times 3000 = 27,492,000$ injections is needed to be done for such a detailed vulnerability analysis. This amount of injection cannot be done in a reasonable time although it is still way smaller than all the possible error candidates (which is $200000 \times 3000 = 600,000,000$). Next section shows some experimental results for MOE increase in SFI when vulnerability factors are calculated for a flip-flop using the same sample for calculating average outcomes.

5.4.3 Vulnerability Factors in RAVEN vs. Error Injection

This section discusses how RAVEN can be employed to design resilient systems. As discussed in previous sections, RAVEN calculates the detection

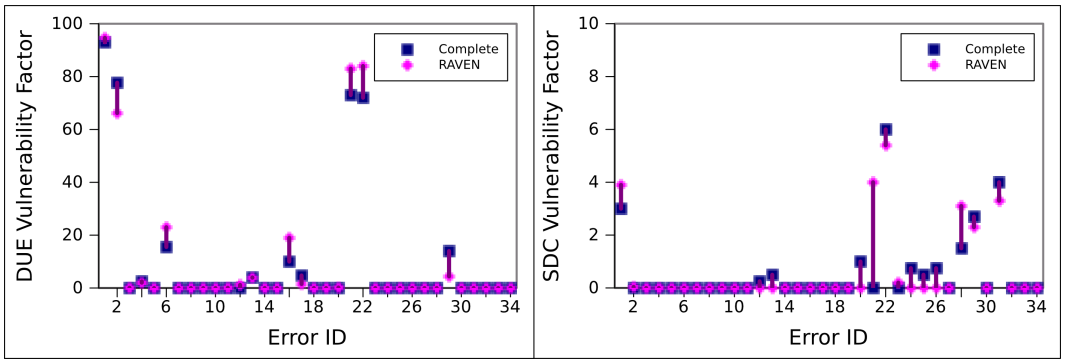


Figure 5.8: RAVEN vs. complete error injection for 400 cycles and 34 random flip-flops.

probability of each flip-flop in the system when a workload is running. These detection probabilities can be interpreted as vulnerability factors calculated in error injection, due to their definition. To find the accuracy of the vulnerability factor for each flip-flop in RAVEN, we need to perform complete error injection for those flip-flops and compare their vulnerability factors with the corresponding ones in RAVEN. Since complete error injection for every possible error candidate and every clock cycle is not feasible, I have performed complete error injection for 34 randomly selected flip-flops during a relatively short period (400 cycles) in the *crafty* workload and calculated RAVEN probabilities for the same time period and the same flip-flops. Then, these two sets of vulnerability factors are compared to each other. This comparison is shown in Fig. 5.8.

As can be seen in this figure, RAVEN probabilities have some inaccuracy due to all the sources of inaccuracy discussed in previous sections. However, in most of the cases, the results of RAVEN and complete error injection are very close, if not equal. Since the complete error injection information is available during this period for the selected flip-flops, vulnerability factors

obtained by SFI with different sample sizes can be simply calculated. This calculation can be done by generating a random list of error candidates for a desired sample size, and selecting the errors injected in the flip-flop of interest. This process has been done for all the confidence levels and MOEs used in Fig. 5.2 for 400 cycles and the vulnerability factors for each sample size has been calculated. Interestingly, these vulnerability factors have a large variance over different sample sizes and in some cases, since there is no error injected on the selected flip-flop, there is no information about the vulnerability factor of that flip-flop. Figure 5.9 shows this variance for some cases. As an example, for a flip-flop with vulnerability factor equal to 78%, sampling calculates vulnerability factors equal to 0%, 85%, 50%, 100%, 80%, 77%, and 75% for sample sizes equal to 23K, 38K, 66K, 93K, 819K, 1.2M, and 1.6M, respectively. For 9.6K and 16.5K samples, there is no vulnerability factor information since that flip-flop was never chosen. The unstable results in SFI is due to a large MOE in vulnerability factor calculation and insufficient sample size per each flip-flop. According to Eq. 5.5, if we want to calculate the vulnerability factors in 400 cycles with MOE=0.05 (i.e., range=0.1), we need 196 samples related to each flip-flop. This results in more than 2.7M injections for the whole IVM.

Due to large MOE of SFI for doable sample sizes, deciding on a resilience technique based on SFI vulnerability factors can result in an over- or under-designed resilient system compared to using vulnerability factors from RAVEN.

5.5 Conclusions and Future Directions

This chapter discussed RAVEN, a new technique for estimating the effects of soft errors for a specified workload. RAVEN achieves significant

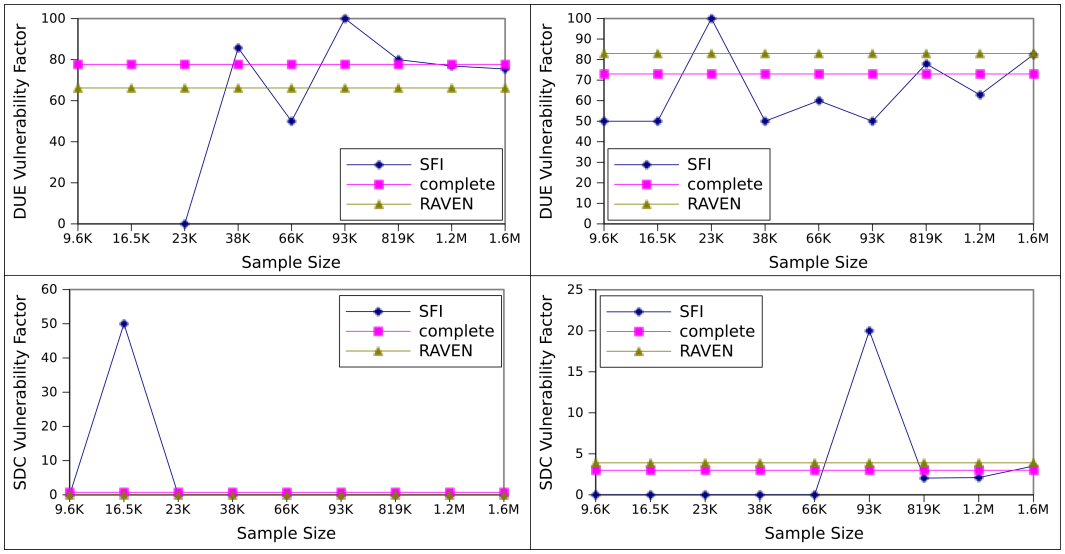


Figure 5.9: Four examples of vulnerability factor values in complete error injection, SFI, and RAVEN.

speedup compared to statistical fault injection, with similar estimated rates of the outcomes and more precise estimated vulnerability factors for each flip-flop.

A future direction that can generate even more precise results is to consider the distribution of errors detected at the local outputs at different times. Using such distributions might even help define lower and upper bounds for RAVEN probability values.

Chapter 6

Conclusions and Future Perspectives

With the increasing size and complexity of VLSI designs, state-of-the-art methodologies for analyzing a design in the presence of faults do not satisfy time-to-market requirements. Statistical methods can estimate the dependability of a design under permanent and transient faults considerably faster than traditional methods. In this dissertation, different statistical methodologies for estimating manufacturing fault coverage and soft error vulnerability of large designs are suggested. They estimate manufacturing fault coverage and soft error vulnerability factors, orders of magnitude faster than current methodologies with low estimation error (less than 10%).

In Chapter 3, a novel metric (GIC) was proposed which was highly correlated with fault coverage, and it was used in Chapter 4 to estimate the fault coverage of a test sequence. An interesting topic for future work is to extend the application of this metric to deal with a subset of faults rather than with the entire set of fault candidates. Such a capability will facilitate performing a hybrid structural and functional test which is an attractive and practical method in industry.

In Chapter 5, it was noted that traditional error injection methodologies require millions of injections in order to be able to calculate detailed vulnerability factors for each flip-flop in the design. Such large numbers of injections are highly time consuming, taking days if not months to complete.

RAVEN, discussed in this chapter, is shown to be effective for such detailed analysis. Another interesting topic for future research is to make this vulnerability analysis even more general. Synthetic benchmarks, rather than generic benchmarks, can be used to measure the vulnerability factors so that they are accurate for any application program. These benchmarks are usually designed to be short and they should be designed in a way to expose the effects of soft errors in every part of the design. Another research that can be very useful in detailed vulnerability analysis is to define new statistical soft error models that can be injected in instruction set simulators or in actual applications [37]. High-level models currently used in soft error analysis are not accurate since their average outcomes are not close to the average outcomes of more realistic (single bit-flip on flip-flops) fault models; furthermore they cannot provide any detailed vulnerability factors for each part of the design. Development of an accurate statistical soft error model has been initiated in [55] and a more precise fault model will enable rapid and accurate analysis of complex systems running realistic workloads.

Bibliography

[1] http://electronicdesign.com/files/29/9345.

[2] "SISA: Simple interactive statistical analysis," http://www.quantitativeskills.com/sisa/statistics/corrhlp.htm.

[3] "Stampede supercomputer," https://www.tacc.utexas.edu/stampede/.

[4] M. Abramovici, M. Breuer, and A. Friedman, *Digital systems testing and testable design*. IEEE press New York, 1990.

[5] V. Agrawal, S. Bose, and V. Gangaram, "Upper bounding fault coverage by structural analysis and signal monitoring," *Proceedings of IEEE VLSI Test Symposium*, pp. 6–pp. IEEE, 2006.

[6] V. D. Agrawal, "Sampling techniques for determining fault coverage in LSI circuits," *Journal of Digital Systems*, vol. 5, no. 3, pp. 189–202, 1981.

[7] V. D. Agrawal, "Fault sampling revisited," *IEEE Design & Test of Computers*, vol. 7, no. 4, pp. 32–35. IEEE, 1990.

[8] G. Asadi and M. B. Tahoori, "An accurate ser estimation method based on propagation probability," *Proceedings of Design, Automation and Test in Europe*, pp. 306–307. IEEE, 2005.

[9] H. Bhatnagar, "Verifault-XL user's guide," *Cadence Design Systems, Inc.*

[10] S. Bose and V. Agrawal, "Estimating stuck fault coverage in sequential logic using state traversal and entropy analysis," *Proceedings of IEEE International Test Conference*, pp. 1–10. IEEE, 2007.

[11] C. Bottoni, M. Glorieux, J. Daveau, G. Gasiot, F. Abouzeid, S. Clerc, L. Naviner, and P. Roche, "Heavy ions test result on a 65nm Sparc-V8 radiation-hard microprocessor," *Proceedings of IEEE International Reliability Physics Symposium, 2014*. IEEE, 2014.

[12] A. Carbine and D. Feltham, "Pentium® Pro processor design for test and debug," *Proceedings of International Test Conference*, pp. 294–303. IEEE, 1997.

[13] H. Cha, E. M. Rudnick, G. S. Choi, J. H. Patel, and R. K. Iyer, "A fast and accurate gate-level transient fault simulation environment," *Digest of Papers: The Twenty-Third International Symposium on Fault-Tolerant Computing*, pp. 310–319. IEEE, 1993.

[14] S. Chatterjee and J. S. Simonoff, *Handbook of Regression Analysis*. John Wiley & Sons, 2013, vol. 5.

[15] C. Chen and N. Soong, "A statistical model for fault coverage analysis," *Digest of Papers of VLSI Test Symposium, 'Chip-to-System Test Concerns for the 90's'*, pp. 227–232. IEEE, 1991.

[16] D. Chen, G. Jacques-Silva, Z. Kalbarczyk, R. K. Iyer, and B. Mealey, "Error behavior comparison of multiple computing systems: A case study using linux on pentium, solaris on SPARC, and AIX on POWER," *Proceedings of Pacific Rim International Symposium on Dependable Computing*, pp. 339–346. IEEE, 2008.

[17] G. Chen, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Object duplication for improving reliability," *Proceedings of Asia and South Pacific Design Automation Conference*, pp. 140–145. IEEE Press, 2006.

[18] L. Chen and S. Dey, "Software-based self-testing methodology for processor cores," vol. 20, no. 3, pp. 369–380, 2001.

[19] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," *Proceedings of ACM/EDAC/IEEE Design Automation Conference*, pp. 1–10. IEEE, 2013.

[20] J. Clary and R. Sacane, "Self-testing computers," *IEEE Computer Magazine*, vol. 12, no. 10, pp. 49–59. IEEE Computer Society, 1979.

[21] H. Cui, S. Seth, and S. Mehta, "Modeling fault coverage of random test patterns," *Journal of Electronic Testing: Theory and Applications*, vol. 19, no. 3, pp. 271–284. Springer, 2003.

[22] J.-M. Daveau, A. Blampey, G. Gasiot, J. Bulone, and P. Roche, "An industrial fault injection platform for soft-error dependability analysis and hardening of complex system-on-a-chip," *Proceedings of International Reliability Physics Symposium*, pp. 212–220. IEEE, 2009.

[23] A. Dharchoudhury, S.-M. Kang, H. Cha, and J. H. Patel, "Fast timing simulation of transient faults in digital circuits," *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pp. 719–722. IEEE Computer Society Press, 1994.

[24] H. Farhat and S. From, "A beta model for estimating the testability and coverage distributions of a VLSI circuit," *IEEE Transactions on*

Computer-Aided Design of Integrated Circuits and Systems, vol. 12, no. 4, pp. 550–554. IEEE, 1993.

[25] M. Fazeli, S. G. Miremadi, H. Asadi, and M. B. Tahoori, "A fast analytical approach to multi-cycle soft error rate estimation of sequential circuits," *Proceedings of Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, pp. 797–800. IEEE, 2010.

[26] G. Ganapathy and J. Abraham, "Hardware acceleration alone will not make fault grading ULSI a reality," *Proceedings of International Test Conference*, p. 848. IEEE, 1991.

[27] P. Goel, "Test generation cost analysis and projections," *Papers on Twenty-five years of electronic design automation*, pp. 380–387. ACM, 1988.

[28] S. Gurumurthy, D. Bertanzetti, P. Jakobsen, and J. Rearick, "Cache-resident self-testing for I/O circuitry," *Proceedings of International Test Conference*, pp. 1–8. IEEE, 2009.

[29] K. Heragu, V. Agrawal, and M. Bushnell, "FACTS: fault coverage estimation by test vector sampling," *Proceedings of VLSI Test Symposium*, pp. 266–271. IEEE, 1994.

[30] K. Heragu, V. Agrawal, and M. Bushnell, "Statistical methods for delay fault coverage analysis," *Proceedings of VLSI Design Conference*, pp. 166–170, 1995.

[31] E. R. Hnatek, *Integrated circuit quality and reliability*. CRC Press, 1994.

[32] D. Holcomb, W. Li, and S. A. Seshia, "Design as you see FIT: System-level soft error analysis of sequential circuits," *Proceedings of the Conference*

on Design, Automation and Test in Europe, pp. 785–790. European Design and Automation Association, 2009.

[33] International Technology Roadmap for Semiconductors, "Test and test equipments," http://www.itrs.net/links/2001itrs/Test.pdf, 2001.

[34] International Technology Roadmap for Semiconductors, "Test and test equipments updates," http://www.itrs.net/links/2012itrs/home2012.htm, 2012.

[35] International Technology Roadmap for Semiconductors, "Test and test equipments," http://www.itrs.net/links/2013ITRS/2013Chapters/2013Test.pdf, 2013.

[36] S. K. Jain and V. D. Agrawal, "STAFAN: An alternative to fault simulation," *Proceedings of the 21st Design Automation Conference*, pp. 18–23. IEEE Press, 1984.

[37] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "FERRARI: A flexible software-based fault and error injection system," vol. 44, no. 2, pp. 248–260. IEEE, 1995.

[38] S. Karthik, M. Aitken, L. Martin, S. Pappula, B. Stettler, P. Vishakantaiah, M. d'Abreu, and J. Abraham, "Distributed mixed level logic and fault simulation on the Pentium® Pro microprocessor," *Proceedings of International Test Conference*, pp. 160–166. IEEE, 1996.

[39] V. Kim, T. Chen, and M. Tegethoff, "Fault coverage estimation for early stage of VLSI design," *Proceedings of Ninth Great Lakes Symposium on VLSI*, pp. 105–108. IEEE, 1999.

[40] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical Fault Injection: quantified error and confidence," *Proceedings of Design, Automation & Test in Europe Conference & Exhibition*, pp. 502–506. IEEE, 2009.

[41] X. Li, S. V. Adve, P. Bose, and J. A. Rivers, "SoftArch: an architecture-level tool for modeling and analyzing soft errors," *Proceedings of International Conference on Dependable Systems and Networks*, pp. 496–505. IEEE, 2005.

[42] R. G. Lomax and D. L. Hahs-Vaughn, *An introduction to statistical concepts, 3rd Ed.* Taylor & Francis Group, 2013.

[43] D. Lyons, "Sun screen," *Forbes, November*, 2000.

[44] H. Ma and A. Sangiovanni-Vincentelli, "Mixed-level fault coverage estimation," *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pp. 553–559. IEEE Press, 1986.

[45] F. J. MacWilliams and N. J. Sloane, "Pseudo-random sequences and arrays," *Proceedings of the IEEE*, vol. 64, no. 12, pp. 1715–1729. IEEE, 1976.

[46] A. Majumdar and S. B. K. Vrudhula, "Fault coverage and test length estimation for random pattern testing," *IEEE Transactions on Computers*, vol. 44, no. 2, pp. 234–247. IEEE, 1995.

[47] R. D. Mason, D. A. Lind, and W. G. Marchal, *Statistics: an introduction*. New York: Harcourt Brace Jovanovich, Inc, 1983.

[48] P. Maxwell, I. Hartanto, and L. Bentz, "Comparing functional and structural tests," *Proceedings of International Test Conference*, pp. 400–407. IEEE, 2000.

[49] P. Maxwell, R. Aitken, V. Johansen, and I. Chiang, "The effect of different test sets on quality level prediction: When is 80% better than 90%," *Proceedings of International Test Conference*, pp. 358–364, 1991.

[50] E. J. McCluskey, "Built-in self-test techniques," *IEEE Design & Test of Computers*, vol. 2, no. 2, pp. 21–28. IEEE, 1985.

[51] W. Meyer and R. Camposano, "Fast hierarchical multi-level fault simulation of sequential circuits with switch-level accuracy," *Proceedings of the 30th international Design Automation Conference*, pp. 515–519. ACM, 1993.

[52] S. Mirkhani and J. A. Abraham, "EAGLE: A regression model for fault coverage estimation using a simulation-based metric," *Proceedings of International Test Conference*. IEEE, 2014.

[53] S. Mirkhani and J. A. Abraham, "Fast evaluation of test vector sets using a simulation-based statistical metric," *Proceedings of VLSI Test Symposium*, pp. 1–6. IEEE, 2014.

[54] S. Mirkhani, J. A. Abraham, T. Vo, H. Jun, and B. Eklow, "FALCON: Rapid statistical fault coverage estimation for complex designs," *Proceedings of International Test Conference*, pp. 1–10. IEEE, 2012.

[55] S. Mirkhani, H. Cho, S. Mitra, and J. A. Abraham, "Rethinking error injection for effective resilience," *Proceedings of Asia and South Pacific Design Automation Conference*, pp. 390–393, 2014.

[56] S. Mirkhani, S. Mitra, C.-Y. Cher, and J. A. Abraham, "Efficient soft error vulnerability estimation of complex designs," *Proceedings of Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2015.

[57] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. S. Kim, "Robust system design with built-in soft-error resilience," *IEEE Computer Magazine*, 2005.

[58] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: An architectural perspective," *Proceedings of International Symposium on High-Performance Computer Architecture*, pp. 243–247. IEEE, 2005.

[59] B. Murray and J. Hayes, "Testing ICs: Getting to the core of the problem," *IEEE Computer Magazine*, vol. 29, no. 11, pp. 32–38, 1996.

[60] A. A. Nair, L. K. John, and L. Eeckhout, "AVF stressmark: Towards an automated methodology for bounding the worst-case vulnerability to soft errors," *Proceedings of 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 125–136. IEEE, 2010.

[61] M. Nakazawa, S. Nitta, and K. Hirabayashi, "Probabilistic fault grading based on activation checking and observability analysis," *Journal of Electronic Testing*, vol. 1, no. 3, pp. 235–238. Springer, 1990.

[62] W. M. Needham, *Designer's guide to testable ASIC devices*. Springer, 1991.

[63] T. Niermann, W. Cheng, and J. Patel, "PROOFS: A fast, memory-efficient sequential circuit fault simulator," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, no. 2, pp. 198–207. IEEE, 1992.

[64] OpenCores, "OpenCores webpage," http://opencores.org/openrisc,OR1200.

[65] P. Parvathala, K. Maneparambil, and W. Lindsay, "FRITS-a microprocessor functional bist method," *Proceedings of International Test Conference*, pp. 590–598. IEEE, 2002.

[66] K. Pattabiraman, G. P. Saggese, D. Chen, Z. Kalbarczyk, and R. Iyer, "Automated derivation of application-specific error detectors using dynamic analysis," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 5, pp. 640–655. IEEE, 2011.

[67] A. Pellegrini, K. Constantinides, D. Zhang, S. Sudhakar, V. Bertacco, and T. Austin, "CrashTest: A fast high-fidelity FPGA-based resiliency analysis framework," *Proceedings of International Conference on Computer Design*, pp. 363–370. IEEE, 2008.

[68] I. Pomeranz, P. K. Parvathala, and S. Patil, "Estimating the fault coverage of functional test sequences without fault simulation," *Proceedings of Asian Test Symposium*, pp. 25–32. IEEE, 2007.

[69] W. Qiu, X. Lu, J. Wang, Z. Li, D. Walker, and W. Shi, "A statistical fault coverage metric for realistic path delay faults," *Proceedings of 22nd IEEE VLSI Test Symposium*, pp. 37–42. IEEE, 2004.

[70] H. M. Quinn, D. A. Black, W. H. Robinson, and S. P. Buchner, "Fault simulation and emulation tools to augment radiation-hardness assurance testing," *IEEE Transactions on Nuclear Science*, vol. 60, no. 3, 2013.

[71] P. Racunas, K. Constantinides, S. Manne, and S. S. Mukherjee, "Perturbation-based fault screening," *Proceedings of International Symposium on High Performance Computer Architecture*, pp. 169–180. IEEE, 2007.

[72] L. K. Rajendra, "Design of on-chip self-testing signature register," Ph.D. dissertation, National Institute of Technology, Rourkela, 2014.

[73] P. Ramachandran, P. Kudva, J. Kellington, J. Schumann, and P. Sanda, "Statistical fault injection," *Proceedings of International Conference on Dependable Systems and Networks With FTCS and DCC*, pp. 122–127. IEEE, 2008.

[74] K. Ramakrishnan, R. Rajaramant, N. Vijaykrishnan, Y. Xie, M. J. Irwin, and K. Unlu, "Hierarchical soft error estimation tool (HSEET)," *Proceedings of International Symposium on Quality Electronic Design*, pp. 680–683. IEEE, 2008.

[75] C. Ravikumar and H. Joshi, "HISCOAP: a hierarchical testability analysis tool," *Proceedings of VLSI Design Conference*, p. 272. Published by the IEEE Computer Society, 1995.

[76] C. Ravikumar, G. Saund, and N. Agrawal, "A STAFAN-like functional testability measure for register-level circuits," *Proceedings of the Fourth Asian Test Symposium*, pp. 192–198. IEEE, 1995.

[77] D. Saab, R. Mueller-Thuns, D. Blaauw, J. Rahmeh, and J. Abraham, "Hierarchical multi-level fault simulation of large systems," *Journal of Electronic Testing*, vol. 1, no. 2, pp. 139–149. Springer, 1990.

[78] P. N. Sanda, J. W. Kellington, P. Kudva, R. Kalla, R. B. McBeth, J. Ackaret, R. Lockwood, J. Schumann, and C. R. Jones, "Soft-error resilience of the IBM POWER6 processor," *IBM Journal of Research and Development*, vol. 52, no. 3, pp. 275–284. IBM, 2008.

[79] N. Seifert, B. Gill, S. Jahinuzzaman, J. Basile, V. Ambrose, Q. Shi, R. Allmon, and A. Bramnik, "Soft error susceptibilities of 22nm tri-gate devices," *IEEE Transactions on Nuclear Science*. IEEE, 2012.

[80] S. Seshu and D. N. Freeman, "The diagnosis of asynchronous sequential switching systems," *IRE Transactions on Electronic computers*, vol. EC-11, pp. 459–465, 1962.

[81] S. C. Seth, V. D. Agrawal, and H. Farhat, "A statistical theory of digital circuit testability," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 582–586. IEEE, 1990.

[82] J. Shen and J. A. Abraham, "Native mode functional test generation for processors with applications to self test and design validation," *Proceedings of International Test Conference*, pp. 990–999. IEEE, 1998.

[83] J. Shen and J. A. Abraham, "Synthesis of native mode self-test programs," *Journal of Electronic Testing*, vol. 13, no. 2, pp. 137–148, 1998.

[84] A. L. Silburt, A. Evans, I. Perryman, S.-J. Wen, and D. Alexandrescu, "Design for soft error resiliency in internet core routers," *IEEE Transactions on Nuclear Science*, vol. 56, no. 6, pp. 3551–3555. IEEE, 2009.

[85] D. T. Smith, B. W. Johnson, N. Andrianos, and J. Profeta III, "A variance-reduction technique via fault-expansion for fault-coverage estimation," *IEEE Transactions on Reliability*, vol. 46, no. 3, pp. 366–374. IEEE, 1997.

[86] B. Swanson and M. Lange, "At-speed testing made easy," http://www.eetimes.com/document.asp?doc_id=1217753, 2004.

[87] R. Taylor, "Interpretation of the correlation coefficient: a basic review," *Journal of diagnostic medical sonography*, vol. 6, no. 1, pp. 35–39. Sage Publications, 1990.

[88] P. Thaker, V. Agrawal, and M. Zaghloul, "Register-transfer level fault modeling and test evaluation techniques for vlsi circuits," *Proceedings of IEEE International Test Conference*, 2000.

[89] T. K. Tsai and R. K. Iyer, *Measuring fault tolerance with the FTAPE fault injection tool*. Springer, 1995.

[90] Univ. of Illinois at Urbana-Champaign, http://www.crhc.illinois.edu/ACS/tools/ivm/about.html.

[91] A. Vij, "Good scan= good quality level? well, it depends," *Proceedings of International Test Conference*, p. 1195. IEEE, 2002.

[92] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel, "Characterizing the effects of transient faults on a high-performance processor pipeline," *Proceedings of International Conference on Dependable Systems and Networks*, pp. 61–70. IEEE, 2004.

[93] Z. Wang and K. Chakrabarty, "Test-quality/cost optimization using output-deviation-based reordering of test patterns," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 2, pp. 352–365. IEEE, 2008.

[94] R. Wieler, Z. Zhang, and R. McLeod, "Emulating static faults using a Xilinx based emulator," *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*. Published by the IEEE Computer Society, 1995.

[95] T. W. Williams, "Test length in a self-testing environment," *IEEE Design & Test of Computers*, vol. 2, no. 2, pp. 59–63. IEEE, 1985.

[96] X. Xu and M.-L. Li, "Understanding soft error propagation using efficient vulnerability-driven fault injection," *Proceedings of 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 1–12. IEEE, 2012.

[97] K. S. Yim, Z. Kalbarczyk, and R. K. Iyer, "Measurement-based analysis of fault and error sensitivities of dynamic memory," *Proceedings of International Conference on Dependable Systems and Networks*, pp. 431–436. IEEE, 2010.

[98] M. Zhang, S. Mitra, T. Mak, N. Seifert, N. J. Wang, Q. Shi, K. S. Kim, N. R. Shanbhag, and S. J. Patel, "Sequential element design with built-in soft error resilience," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 12, pp. 1368–1378. IEEE, 2006.

[99] Y. Zhang, J. W. Lee, N. P. Johnson, and D. I. August, "DAFT: decoupled acyclic fault tolerance," *International Journal of Parallel Programming*, vol. 40, no. 1, pp. 118–140. Springer, 2012.

Vita

Shahrzad Mirkhani was born in Tehran, Iran in 1976, daughter of Javad Mirkhani and Mahin Safaei Tehrani. She earned B.Sc. degree in Computer Engineering from Sharif University of Technology, Tehran, Iran, in 1998. She received her M.Sc. degree in Electrical and Computer Engineering from University of Tehran, Tehran, Iran, in 2002. Shahrzad has worked as a researcher in CADLAB in University of Tehran between 2002 and 2007. She has been a graduate student at the University of Texas at Austin since 2008.

Permanent address: shahrzad@utexas.edu

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.