

**DESIGN-FOR-TESTABILITY AND DIAGNOSIS
METHODS TO TARGET UNMODELED
DEFECTS IN INTEGRATED CIRCUITS AND
MULTI-CHIP BOARDS**

by

Hongxia Fang

Department of Electrical and Computer Engineering
Duke University

Date: _____

Approved:

Krishnendu Chakrabarty, Supervisor

Chris Dwyer

Gershon Kedem

Hisham Z. Massoud

Xinli Gu

Dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Electrical and Computer Engineering
in the Graduate School of
Duke University

2011

ABSTRACT

**DESIGN-FOR-TESTABILITY AND DIAGNOSIS
METHODS TO TARGET UNMODELED
DEFECTS IN INTEGRATED CIRCUITS AND
MULTI-CHIP BOARDS**

by

Hongxia Fang

Department of Electrical and Computer Engineering
Duke University

Date: _____

Approved:

Krishnendu Chakrabarty, Supervisor

Chris Dwyer

Gershon Kedem

Hisham Z. Massoud

Xinli Gu

An abstract of a dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Electrical and Computer Engineering
in the Graduate School of
Duke University

2011

Copyright © 2011 by Hongxia Fang
All rights reserved

Abstract

Very deep sub-micron process technologies are leading to increasing defect rates for integrated circuits (ICs) and multi-chip boards. To ensure the quality of test patterns and more effective defect screening, functional tests, delay tests, and n -detect tests are commonly used in industry for detecting unmodeled defects. However, the resulting test data volume and test application time are prohibitively high. Moreover, functional tests suffer from low defect coverage since they are mostly derived in practice from existing design-verification test sequences. Another challenge is that it is hard to find the root cause for board-level functional failures caused by various types of unmodeled defects. Therefore, there is a need for efficient testing, design-for-testability (DFT), and fault diagnosis methods to target these unmodeled defects.

To address the problem of high test data volume, a number of test compression methods have been proposed in the literature based on Linear-Feedback-Shift-Register (LFSR) reseeding. A seed can be computed for each test cube by solving a system of linear equations based on the feedback polynomial of the LFSR. To improve the effectiveness of these seeds for unmodeled defects, this thesis shows how the recently proposed output-deviations metric can be used to select appropriate LFSR seeds. To further reduce the requirement for automatic test equipment (ATE) memory, the thesis describes a DFT technique for increasing the effectiveness of LFSR reseeding for unmodeled defects, which relies on seed selection using the output-deviations metric and the on-chip augmentation of seeds using simple bit-operations.

Functional test sequences are often used in manufacturing testing to target defects that are not detected by structural test. In order to reduce the test application time caused by long functional test, which is needed to obtain high defect coverage, a deviation-based method to grade functional test sequences at register transfer

(RT)-level is presented. The effectiveness of the proposed method is evaluated by the correlation between the RT-level deviations and gate-level fault coverage. It is also evaluated on the basis of coverage ramp-up data. Since functional test usually suffer from low defect coverage, there is a need to increase their effectiveness using DFT techniques. A DFT method has therefore been developed—it uses the register-transfer level (RTL) output-deviations metric to select observation points for an RTL design and a given functional test sequence.

A common scenario in industry is “No Trouble Found” (NTF), which means that each chip on the board passes the ATE test while it fails during board-level test. In such case, it is necessary to perform diagnosis at board-level to find the root cause. A promising approach to address this problem is to carry out fault diagnosis in two phases—suspect faulty components on the board or modules within components are first identified and ranked, and then fine-grained diagnosis is used to target the suspect blocks in ranked order. In this thesis, a new method based on dataflow analysis and Dempster-Shafer theory is presented for ranking faulty blocks in the first phase of diagnosis.

To find the root cause of NTF, an innovative functional test approach and DFT methods have been developed for the detection of board-level functional failures. These DFT and test methods allow us to reproduce and detect functional failures in a controlled deterministic environment, which can provide ATE tests to the supplier for early screening of defective parts.

In summary, this research is targeted at the testing, DFT, and diagnosis of unmodeled defects. The proposed techniques are expected to provide high-quality and compact test patterns, and effective DFT methods for various types of defects in integrated circuits. It is also expected to provide accurate diagnosis to find the root cause of defects in multi-chip boards.

Acknowledgements

I would like to thank several people whose help and contribution has made the completion of this dissertation possible.

First I would like to thank my advisor, Prof. Krishnendu Chakrabarty, for his continuous guidance, support and inspiration throughout this work. His technical insights, insistence on high-quality work and encouragement on my progress have influenced me a lot. I learned from him on how to identify the key issues in VLSI testing and diagnosis field, how to model and state the existing problems, how to find the promising ideas and effective solutions through analytical thinking, how to enhance the proposed method and so on. All these training has equipped me with necessary skills and qualities to be a great researcher.

I am grateful to my committee members Prof. Hisham Z. Masoud, Prof. Chris Dwyer, Prof. Gershon Kedem, and Dr. Xinli Gu for their time and their valuable suggestions and feedback on my work.

I would like to thank my industry mentors and cooperators. Dr. Xinli Gu, former director in Cisco, has provided me multiple internship opportunities in his group and I appreciate it very much. These internship experience has benefited me a lot in both my research and my industry skills. I would also like to thank Dr. Brian Wang, technical leader in Cisco, for his direct help and mentor. He has provided industry-insights on the board-level functional failure diagnosis project as well as guiding my research on this topic. I thank Rubin Parekhji of TI Corporation for his constructive suggestion on the seed augmentation work. I thank Dr. Abhijit Jas, Dr. Srinivas Patil and Dr. Chandra Tirumurti of Intel Corporation for their effort on the functional test sequences grading work.

I would like to thank people in my research group Zhanglei Wang, Mahmut Yil-

maz, Sudarshan Bahukudumbi, Tao Xu, Yang Zhao, Zhaobo Zhang and Brandon Noia. I have benefited greatly from the friendly and active office environment.

I am grateful for financial support I received for my graduate studies from the Semiconductor Research Corporation and Cisco Systems, Inc.

Finally, I would like to thank my parents for their consistent support for my studies. A special thanks goes to my husband, Chengzhi Li. He has always been there with me to share my pains and joys. His love and support has encouraged me to overcome difficulties in both my life and research. This dissertation will not be completed so smoothly without the support and accompaniment he has provided over the years.

Contents

Abstract	iv
Acknowledgements	vi
List of Figures	xii
List of Tables	xvii
1 Introduction	1
1.1 Background	1
1.1.1 Manufacturing Testing and Verification Testing	2
1.1.2 Functional Test and Structural Test	2
1.1.3 Fault Models	4
1.1.4 Test Generation	6
1.1.5 Design for Testability	7
1.1.6 Test Data Compression	8
1.1.7 Diagnosis	9
1.2 Motivation for Thesis Research	10
1.2.1 LFSR Reseeding Targeting Unmodeled Defects	11
1.2.2 Seed Augmentation for LFSR Reseeding	12
1.2.3 Functional Test Grading at RT-Level	13
1.2.4 RTL DFT for Functional Test Sequences	15
1.2.5 Ranking Candidate Blocks for Diagnosis of Board-Level Functional Failures	17
1.2.6 Design-for-Testability and Diagnosis Methods for Board-Level Functional Failures	20

1.3	Thesis Outline	22
2	Seed Selection and Seed Augmentation in Test Compression	26
2.1	Probabilistic Fault Model and Output Deviations	26
2.2	LFSR Reseeding Targeting Unmodeled Defects	28
2.2.1	Seed Selection	29
2.2.2	Experimental Evaluation	35
2.3	Seed Augmentation for LFSR Reseeding	53
2.3.1	Bit-Operations-Based Seed Augmentation	53
2.3.2	Hardware Implementation	55
2.3.3	Experimental Evaluation	60
2.4	Chapter Summary and Conclusions	66
3	RTL Grading and RTL DFT for Functional Test Sequences	68
3.1	Output Deviations at RT-level: Preliminaries	68
3.2	Deviation Calculation at RT-level	69
3.2.1	Observability Vector	70
3.2.2	Weight Vector	71
3.2.3	Threshold value	73
3.2.4	Calculation of Output Deviations	75
3.3	Functional Test Grading at RT-Level	77
3.3.1	Experimental Evaluation	77
3.4	RTL DFT for Functional Test Sequences	101
3.4.1	Problem Formulation	101
3.4.2	Observation-Point Selection	102
3.4.3	Experimental Evaluation	107

3.5	Chapter Summary and Conclusions	116
4	Ranking Candidate Components/Blocks for Diagnosis of Board-Level Functional Failures	117
4.1	Dempster-Shafer Theory and Application	118
4.2	Ranking Candidate Blocks Based on Beliefs	122
4.2.1	Stage partitioning for the given functional test	123
4.2.2	Dataflow extraction	124
4.2.3	Assignment of belief based on each failing stage	126
4.2.4	Combination of beliefs and ranking of blocks	128
4.3	Experimental Evaluation	129
4.3.1	Overview of <i>design_A</i>	130
4.3.2	Simulation results	130
4.4	Chapter Summary and Conclusions	135
5	Design-for-Testability and Diagnosis Methods for Board-Level Functional Failures	136
5.1	Motivation and Problem Statement	136
5.2	Functional Scan Design and Functional Scan Test	138
5.2.1	Functional scan design	138
5.2.2	Functional scan test	141
5.3	Key Definitions	143
5.4	Reproduction and Detection of Board-Level Functional Failures	145
5.5	Experimental Evaluation	147
5.5.1	Experimental setup	147
5.5.2	Mimicking of functional state space	148

5.5.3	Reproducing and detecting functional failures	153
5.5.4	Functional failures due to more random faults	155
5.6	Automated Extraction on Functional Constraints	156
5.7	Techniques to Mimic Functional State Space	158
5.7.1	Multiple initial states	159
5.7.2	State-injection technique	163
5.7.3	Experimental results	166
5.8	Chapter Summary and Conclusions	170
6	Conclusions and Future Work	171
6.1	Thesis Contributions	172
6.2	Future Work	174
6.2.1	Diagnosis Based on a Fault Model	174
6.2.2	Additional Diagnostic Test Generation	175
6.2.3	Fault Diagnosis for Designs with Communication to Memory .	177
	Bibliography	178
	Biography	186

List of Figures

1.1	Flip-flops in a circuit connected as a scan chain.	7
1.2	A generic BIST architecture for scan designs.	8
1.3	Test data compression.	9
1.4	Blocks (components and modules within components) on a board. . .	20
2.1	An example to illustrate confidence levels.	27
2.2	(a) An LFSR and phase shifter; (b) State transition matrix of the LFSR.	29
2.3	Example to illustrate the solution for a system of linear equations. . .	31
2.4	Procedures: (a) LFSR reseeding with seed selection, (b) Sort test patterns according to output deviations.	32
2.5	An example for the sorting of test patterns based on output deviations.	32
2.6	Applying a test pattern pair using LOS and LOC schemes.	34
2.7	Fault coverage for s5378 and s9234 (5-detect stuck-at test cubes): (1) stuck-at faults; (2) stuck-open faults (LOS); (3) transition faults (LOC).	46
2.8	Fault coverage for s15850 and s38417 (5-detect stuck-at test cubes): (1) stuck-at faults; (2) stuck-open faults (LOS); (3) transition faults (LOC).	47
2.9	Fault coverage for tv80 and mem_ctrl (1-detect stuck-at test cubes): transition faults (LOC, p_1).	48
2.10	Fault coverage for ac97_ctrl and DMA (1-detect stuck-at test cubes): transition faults (LOC, p_1).	48
2.11	Fault coverage for pci_bridge32 and ethernet (1-detect stuck-at test cubes): transition faults (LOC, p_1).	49

2.12	Fault coverage for tv80 and mem_ctrl (3-detect stuck-at test cubes): transition faults (LOC, p_1).	49
2.13	Fault coverage for ac97_ctrl and DMA (3-detect stuck-at test cubes): transition faults (LOC, p_1);	50
2.14	Fault coverage for pci_bridge32 (3-detect stuck-at test cubes): transition faults (LOC, p_1).	50
2.15	Fault coverage for tv80 and mem_ctrl (1-detect transition test cubes): (1) stuck-at faults; (2) transition faults (LOC, p_2).	51
2.16	Fault coverage for DMA and pci_bridge32 (1-detect transition test cubes): (1) stuck-at faults; (2) transition faults (LOC, p_2).	52
2.17	Architecture for bit-operation-based LFSR reseeding.	56
2.18	State transition diagram of the FSM.	58
2.19	Logic design for Seed Generator (example of 4-bit seeds).	59
2.20	Fault coverage ramp-up for s38417, s38584 and tv80 for Experiment 1: (1) transition faults (LOC); (2) stuck-at faults and bridging faults.	60
2.21	LOC transition fault coverage ramp-up for Experiment 3: (1) s38417; (2) tv80.	61
2.22	LOC transition fault coverage ramp-up for Experiment 3: (1) ac97_ctrl; (2) DMA.	61
2.23	Fault coverage ramp-up for DMA for Experiment 1: (1) transition faults (LOC); (2) stuck-at faults and bridging faults.	61
2.24	Fault coverage ramp-up for s38417, tv80 and DMA for Experiment 2: (1) transition faults (LOC); (2) stuck-at faults and bridging faults.	65
2.25	LOC transition fault coverage ramp-up for Experiment 3: (1) s38417; (2) tv80; (3) DMA.	66
3.1	(a) Architecture of the Parwan processor; (b) Dataflow graph of the Parwan processor (only the registers are shown).	71

3.2	An example to illustrate cumulative new transition counts.	74
3.3	Cumulative new transition counts for T_1 and T_4 (Biquad filter).	80
3.4	Experimental flow.	80
3.5	Correlation between output deviations and gate-level fault coverage (Biquad).	81
3.6	Correlation between output deviations (only consider transition counts) and gate-level fault coverage (Biquad).	82
3.7	Cumulative stuck-at fault coverage for various test-sequence ordering methods (Biquad).	83
3.8	Cumulative BCE+ for various test-sequence ordering methods (Biquad).	83
3.9	Correlation result for longer functional test sequences (Biquad).	84
3.10	Dataflow diagram of Scheduler module.	87
3.11	Cumulative new transition counts for T_0 and T_1 (Scheduler module).	88
3.12	Cumulative new transition counts for T_6 and T_7 (Scheduler module).	89
3.13	Correlation between output deviations and gate-level fault coverage (Scheduler module).	91
3.14	Correlation between output deviations (only consider transition counts) and gate-level fault coverage (Scheduler module).	91
3.15	Cumulative stuck-at fault coverage for various test-sequence ordering methods (Scheduler module).	92
3.16	Cumulative transition fault coverage for various test-sequence ordering methods (Scheduler module).	92
3.17	Cumulative BCE+ metric for various test-sequence ordering methods (Scheduler module).	93

3.18	Correlation between output deviations and gate-level fault coverage for <i>imp2</i> (Scheduler module).	97
3.19	Correlation between output deviations and gate-level fault coverage for <i>imp3</i> (Scheduler module).	98
3.20	Correlation results for more functional test sequences (Scheduler module).	101
3.21	Results on gate-level normalized stuck-at fault coverage.	112
3.22	Results on gate-level normalized transition fault coverage.	113
3.23	Results on the gate-level normalized <i>BCE+</i> metric.	114
3.24	Results on gate-level normalized GE score.	115
4.1	An example of a typical functional test sequence.	123
4.2	Dataflow graph for: (a) <i>stage1</i> ; (b) <i>stage2</i> ; (c) <i>stage3</i>	126
4.3	Belief assignment based on <i>stage1</i> of the given functional test.	129
5.1	An illustration of the state space for different test methods.	137
5.2	An example of functional scan design.	139
5.3	An example of clock timing for functional scan test.	142
5.4	Examples for extracting assignment conditions.	149
5.5	An example of substate group partitioning.	149
5.6	Distribution of the number of unique functional states (<i>design_A</i>).	151
5.7	Distribution of the number of unique functional state transitions (<i>design_A</i>).	152
5.8	Distribution of the number of unique functional state sequences of depth 3 (<i>design_A</i>).	152

5.9	Reproduction and detection results for functional failures due to more random faults.	156
5.10	Motivation for the use of multiple initial states.	159
5.11	An example of weighted state transition graph G	162
5.12	Graph G' for strong connectivity components.	162
5.13	Comparison of $Cov_SS_k_α(k = 3, α = 0.8)$ for with and without state-injection (<i>design_A</i>).	169
5.14	Comparison of $Cov_SS_k_α(k = 3, α = 0.85)$ for with and without state-injection (<i>design_A</i>).	169
5.15	Comparison of $Cov_SS_k_α(k = 3, α = 0.9)$ for with and without state-injection (<i>design_A</i>).	169
6.1	The diagnosis flow of functional failures.	175
6.2	Flow for constrained ATPG and validation.	176
6.3	Validation example.	177

List of Tables

2.1	Signal probabilities for different input combinations.	27
2.2	The various seed sets obtained using pattern pairs (p_1, p_2)	34
2.3	Comparison of compression results.	38
2.4	Selection of p_1 or p_2 in calculation of deviations.	39
2.5	Characteristics of IWLS-05 benchmarks.	41
2.6	Characteristics of n -detect stuck-at test cubes.	43
2.7	1-detect LOC transition-fault test cubes.	44
2.8	Interface of the FSM.	57
2.9	States of the FSM.	58
2.10	Comparison of storage requirement (number of seeds).	63
3.1	Weight vector for registers (Parwan).	72
3.2	Registers and register arrays in the Scheduler module.	85
3.3	Weight vector for Scheduler module.	86
3.4	RT-level deviations of ten functional tests for Scheduler module. . . .	90
3.5	Length of LCS.	94
3.6	Comparison of three implementations.	95
3.7	RT-level deviations of ten functional tests for <i>imp2</i> of Scheduler module. . .	96
3.8	RT-level deviations of ten functional tests for <i>imp3</i> of Scheduler module. . .	97
3.9	Correlation results with stratified sampling (Scheduler module). . . .	99

3.10	Internal effective TCs for test sequence TS	104
3.11	Gate-level fault coverage (stuck, transition and $BCE+$) of the design before and after inserting all observation points.	110
3.12	Gate-level GE score of the design before and after inserting all observation points.	110
4.1	Combination of belief measures.	121
4.2	Toggled characteristic signals in time slots of each stage.	125
4.3	Stages of the given functional test for $design_A$	131
4.4	Results on ranking for failures due to the injected faults.	132
5.1	Extracted assignment conditions for registers from Fig. 5.5.	150
5.2	α -coverage metrics for functional scan test for $design_A$	153
5.3	Functional failure detection using functional scan test for $design_A$	155
5.4	α -coverage metrics for functional scan test with improved functional constraints.	158
5.5	Functional failure detection using functional scan test with improved functional constraints.	158
5.6	Comparison of α -coverage metrics for the proposed initial-state selection method and random method for $design_A$	168

Chapter 1

Introduction

The testing of an integrated circuit (IC) or board is the process of exercising it with test patterns and analyzing its responses to determine whether it behaves correctly. Once an IC or board is fabricated, it is subjected to manufacturing testing to prevent the delivery of defective parts to customers. For each failed chip/board, a diagnosis procedure must be used to find the root cause for these failures. The diagnosis procedure is used to identify and locate the faulty logic in the IC/board.

Shrinking process technologies and higher design complexity are leading to increasing defect rates. Therefore, the focus of this research is on manufacturing testing and diagnosis of ICs and boards targeting unmodeled defects. Specific techniques are proposed for developing high-quality and low-cost test solutions, and accurate diagnosis procedures are described for high-density nanometer ICs/boards.

The rest of this chapter is organized as follows. Section 1.1 introduces background material on very large scale integrated (VLSI) circuit testing and diagnosis. Section 1.2 describes the motivation for this research and the related previous work. The outline of this document is presented in Section 1.3.

1.1 Background

In this section, we briefly introduce the key definitions and terminology that are widely used in the testing and diagnosis field. Specially, we present the relationship between manufacturing testing and verification testing, describe functional test and structural test, and introduce fault models [1], test-data compression, and diagnosis.

1.1.1 Manufacturing Testing and Verification Testing

The testing of an VLSI circuit is a process to determine whether the circuit behaves correctly by applying a test stimulus to the circuit and analyzing the responses. Testing can be classified into four categories based on its purpose: verification testing, manufacturing testing, burn-in, and incoming inspection [2]. In this document, we clarify the verification testing and manufacturing testing, since these terms are often used incorrectly in the literature.

Verification testing is also known as characterization or silicon debug. It is performed on a new design before it is sent for mass production. The purpose is to verify the correctness of the design and the test procedure, to ensure that the design meets all the requirements of the specification, and to diagnose and correct the possible errors in the design and in the test procedure. It continues throughout the production life of chips to improve the design and the process to increase yield.

Manufacturing testing is performed at the factory for testing of all manufactured chips for parametric faults, random, and systemic defects. The purpose of manufacturing testing is to determine whether manufactured chips in mass production meet specifications. Generally, it must cover a high proportion of modeled faults to ensure test quality, and it must minimize the test time to control test cost. It only decides whether the target chip is good or bad and includes no fault diagnosis procedure. Every fabricated chip is subject to manufacturing testing, such that defective chips are not delivered to customers.

In this research, we address the problem of manufacturing testing.

1.1.2 Functional Test and Structural Test

The main objective of functional test [2] is to verify the correctness of a circuit with respect to its functional specifications. Functional test is usually derived from the

functional model that reflects the function of the circuit and it is independent of the specific implication.

The simplest functional test is the exhaustive test. It detects almost any fault by applying stimulus of all possible combinations to primary inputs. Due to the length of the resulting tests, exhaustive testing can only be feasible for small circuits. In practice, the functional test targets only a subset of all the design's functions, usually the critical functions identified by designers. Most functional tests used during manufacturing testing are generated from the tests used in the simulation-based verification process.

A major problem with functional testing is that it lacks adequate ways to evaluate the effectiveness of these tests. Software test coverage criteria, such as statement coverage, branch coverage, and path coverage have been used for evaluation. However, they are not adequate to determine how many faults or defects are detected.

The advantages of functional tests are the following:

- Functional tests can be derived without information about low-level designs;
- Patterns used in the design verification process can be transformed into functional tests, which greatly reduce the effort for developing tests;
- Functional tests can detect some hard-to-detect defects.

However, functional testing also suffers from some drawbacks. These include:

- The length of a functional test is usually prohibitively long;
- The defect coverage provided by functional tests is low;
- There is no efficient way to evaluate the effectiveness of a functional test.

In contrast to functional testing, structural testing [2] targets the structure of the circuit, which is related to the actual physical defects. It relies on specific structural fault models. Details about fault models can be found in the following subsection. Structural tests can be generated only if the gate-level netlist for the target circuit is available. Usually, a short test time is sufficient to achieve high fault coverage for a given fault model. Commonly used fault models are the stuck-at fault model, the delay fault model, the bridging fault model and the crosstalk fault model.

The advantages of structural testing are as follows:

- High fault coverage can be achieved using structural tests;
- Usually the length of the structural test is short;
- The structural test can be properly evaluated for fault coverage based on various fault models.

The disadvantages of structural testing are as follows:

- A gate-level netlist is needed for structural tests;
- Yield loss due to overtesting is inevitable in structural testing;

1.1.3 Fault Models

Fault models [2] characterize the manifestation physical defects in circuits. A fault model is a set of logical faults that represent the effect of the physical defects on the behavior of the modeled system. Testing involves the detection of faults from a given fault model [1]. With fault models, the analysis and detection of physical defects becomes a logical problem. This makes automatic test generation feasible. Various fault models have been developed to model different kind of defects. This subsection presents three widely used fault models: stuck-at fault model, transition fault model, bridge fault model.

Stuck-at Fault Model

The stuck-at fault model [2] is the most commonly used fault model in industry. It assumes that every faulty line is either permanently set to the logic value 0 or logic value 1, corresponding to s-a-0 or s-a-1 fault. According to the single-fault assumption, there can be up to $2n$ single stuck-at faults for a circuit with n lines. This number can be reduced by using fault-collapsing techniques. A test set derived based on the stuck-at fault model has been shown to be effective for detecting many defects during manufacturing test.

Transition Fault Model

Many defects and failures are related to timing issues for today's deep sub-micron technologies. The transition fault model [2] was proposed to deal with the timing defects. It is based on the assumption that the delay fault affects only one gate in the circuit and it causes the combinational delay of the circuit to exceed the clock period. Each gate can be affected by two possible transition faults: slow-to-rise and slow-to-fall. Since the sites of transition faults are the same as that of stuck-at faults, the number of transition faults is also $2n$ for a circuit with n lines. Also, the tools for generating tests targeting stuck-at faults can be easily modified to generate transition fault tests.

Bridging Fault Model

A bridging fault represents a short between a group of signals [2]. There are usually two types of bridging faults: OR-type bridging and AND-type bridging. In OR-type bridging, the logic value of the shorted net is 1-dominant, while the logic value of the shorted net is 0-dominant in the AND-type bridge. Bridging faults are often used as

examples of “defect-oriented faults” [3].

Based on the given fault model, fault coverage can be obtained by running fault simulation. Fault coverage is a metric for evaluating the effectiveness of a given test set. Suppose that the total number of faults in a circuit for the targeted fault model is n , and suppose a test set can detect m out of these n faults. Then the fault coverage is $m/n \times 100\%$. In this thesis, stuck-at, transition, and bridging faults are considered.

1.1.4 Test Generation

Test generation (TG) is the process of determining the input test patterns necessary to test a CUT. TG can be fault-oriented or function-oriented. Fault-oriented TG generates test patterns that detect faults defined by specific fault models. Function oriented TG generates test patterns that are intended to show that the system performs its specified function [1]. Automatic test pattern generation (ATPG) is the process of automatically generating test patterns for the detection of a specific fault group using an algorithm.

The terms controllability [4] and observability [5] are commonly used in ATPG literature. Controllability is a metric to measure the difficulty of setting a node to a specific logic value. As we go deep into the circuit, the controllability of the circuit becomes more difficult. Observability is a metric to measure the difficulty in propagating a signal value to primary outputs or test observation points.

Many companies today rely on n -detect [6] ATPG for more effective defect screening. The n -detect ATPG flow is similar to the basic ATPG flow. Since n -detect tests are usually developed based on the single stuck-at fault model, existing stuck-at fault ATPG tools can be used with little modifications. The difference is that each fault is aimed to be detected at least n times before dropping it from the overall fault list. The value added by n -detect ATPG is the efficiency in detecting unmodeled defects

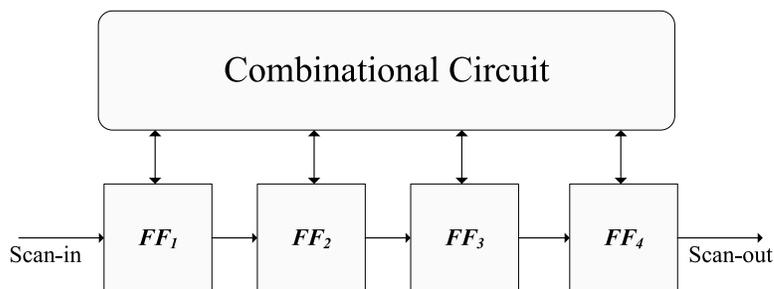


Figure 1.1: Flip-flops in a circuit connected as a scan chain.

that would otherwise be missed by basic ATPG flows [7]. The drawbacks are the increased test pattern count and the longer pattern generation time.

1.1.5 Design for Testability

The purpose of design for testability (DFT) is to make the testing of the CUT easier and more efficient. The most commonly used DFT techniques are scan design, built-in self-test (BIST). In this section, we will briefly discuss these techniques.

Scan Design

Scan design [2] is a widely used DFT technique that provides controllability and observability for flip-flops by adding a scan mode to the circuit. When the circuit is in scan mode, all the flip-flops form one or more shift registers, also known as scan chains. Using separate scan access I/O pins, test patterns are serially shifted in to the scan chains and test responses are serially shifted out. This process significantly reduces the cost of test by transforming the sequential circuit into a combinational circuit for test purposes. For circuits with scan designs, the test process involves test pattern application from external ATE to the primary inputs and scan chains of the DUT. To make a pass/fail decision on the DUT, the states of the primary outputs and the flip-flops are fed back to the ATE for analysis. Figure 1.1 illustrates how flip-flops are connected to form a scan chain.

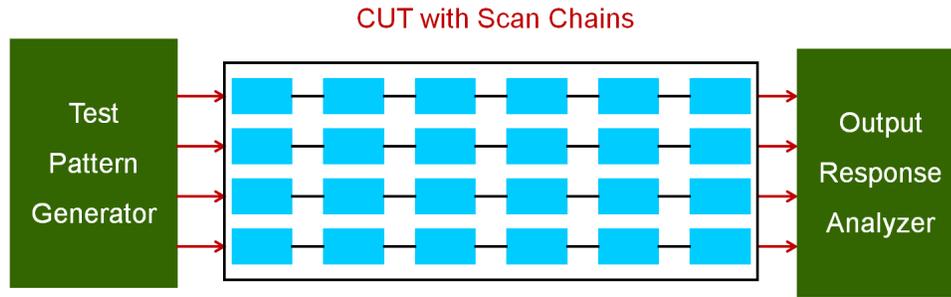


Figure 1.2: A generic BIST architecture for scan designs.

Built-in Self-test

Built-in self-test (BIST) is one of the most popular DFT techniques and it is widely used in industry. BIST removes the need of expensive external testers, allows at-speed testing, and alleviates the limitation on the number of tester channels. In BIST architecture, test patterns are generated by an on-chip test pattern generator. These generated patterns are then fed to scan chains. After capture cycles, the values of the scan cells are fed to the output response analyzer. For scan designs, the test pattern generator is usually named pseudo-random-pattern-generator (PRPG), which is a linear-feedback shift-register (LFSR) that generates pseudo-random patterns. Figure 1.2 illustrates a generic BIST architecture for scan designs.

1.1.6 Test Data Compression

To ensure the high defect coverage for large circuits, we require large test data volume, and thus large tester memory and long test application time. To address the problem of excessive test data volume, various test data compression techniques have been developed [8] [9] [10]. The basic idea of test data compression is to compress the test vectors before test application and decompress them for the circuit-under-test (CUT) during test application. The response of the CUT can also be compressed before it is sent to the ATE. Figure 1.3 shows an example for test data compression. A test

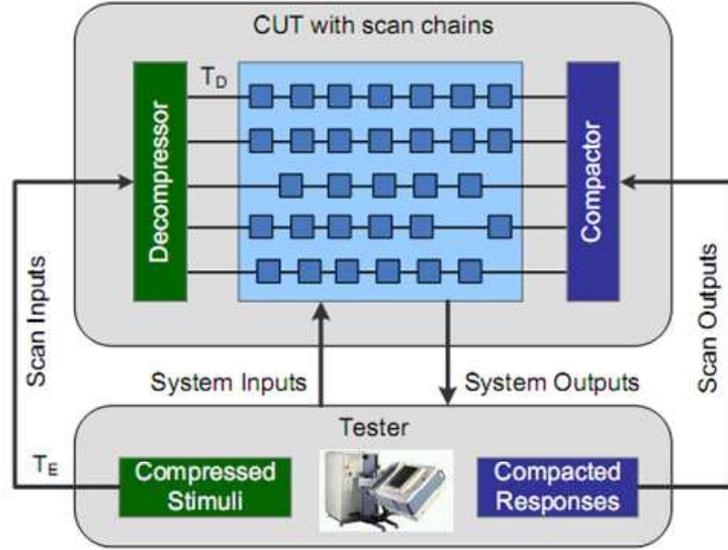


Figure 1.3: Test data compression.

set T_D for the CUT is compressed to a much smaller data set T_E , which is stored in ATE memory. An on-chip decompressor is used to generate T_D from T_E during test application.

1.1.7 Diagnosis

If a chip/board fails a test in manufacturing testing, it is essential to find the root cause for this failure. The goal of diagnosis is to identify and locate the circuit logic that leads to the failure. Conventionally, diagnosis techniques can be divided into two categories: fault dictionary based methods [11] [12], and effect-cause reasoning methods [13] [14]. The fault-dictionary-based methods use simulation to characterize the behavior of circuit for all faults. These methods become infeasible when the size of the fault-list grows too large. The effect-cause reasoning methods study the results of test application and deduce the possible reasons for the failures.

1.2 Motivation for Thesis Research

Due to shrinking process technologies, many new defect types, which cannot be modeled using existing fault models, cause failure in chips/boards. To ensure the quality of test patterns, at-speed functional testing, delay-test methods, and n -detect test sets are commonly used in industry for detecting unmodeled defects. However, the resulting test data volume is often prohibitively high. It was recently predicted that the test data volume for integrated circuits will be as much as 38 times larger and the test application time will be about 17 times larger in 2015 than they were in 2007 [15].

In order to solve these problems, we have proposed a LFSR reseeding-based test compression method to target high defect coverage with a small number of seeds stored on the ATE. To further reduce memory requirement with no adverse impact on defect coverage, we have proposed bit-operation-based seed-enrichment method to generate new seeds on-chip. Since functional tests are commonly used in industry and they also lead to high test data volume and long test application time, we have developed a method to grade functional test sequences at RT-level and only select a subset of the entire test sequences for use in manufacturing test. To increase the defect coverage of an existing functional test sequence TS , we have proposed RTL DFT techniques to enhance the testability of the design, thus making better use of TS for targeting defects in manufacturing testing.

Another challenge today is the diagnosis of functional failures at board-level. We have proposed a method based on dataflow analysis and Dempster-Shafer (DS) theory for ranking candidate blocks in the diagnosis of board-level functional failures. We have also proposed DFT and diagnosis flow to allow reproduce and detect board-level functional failures in a controlled, deterministic test environment.

The thesis research is focused on the topics mentioned above: LFSR reseeding,

seed augmentation, functional test grading, RTL DFT, candidate blocks ranking, DFT and diagnosis methods for board-level functional failures. The motivation and related previous work for these problems are detailed below.

1.2.1 LFSR Reseeding Targeting Unmodeled Defects

Test compression is essential to reduce test data volume and testing time. A category of test-compression techniques relies on the use of a linear decompressor such as an LFSR or an XOR network. These methods exploit the fact that test cubes for modeled faults in full-scan circuits contain very few specified (care) bits; a test cube with many don't-care bits can be easily generated using an LFSR with a compact seed or by using a simple XOR “expander” network. Such compression techniques have been implemented in commercial tools such as TestKompres [16].

LFSR reseeding has long been recognized as an efficient test compression technique [17], [18], [19], [20]. A test cube, whose length equals the number of controllable inputs (primary inputs and scan cells) in the circuit, can be encoded with high probability into a compact seed of typical length $S_{max} + 20$, where S_{max} equals the number of specified bits in the test cube [17]. A seed can be computed for each test cube by solving a system of linear equations based on the feedback polynomial of the LFSR. The solution space for the system of linear equations is quite large, especially for test cubes with few specified bits. For example, if the LFSR has n stages and the number of care bits in the test cube is m , the number of solutions is at least 2^{n-m} . All patterns derived from the possible solutions (LFSR seeds) cover the original test cube. However, these patterns differ from each other in their ability to detect unmodeled defects. Therefore, it is important to select LFSR seeds that can be used to target a larger number of defects. Moreover, if more effective seeds are loaded first into the LFSR, a steeper ramp-up of fault coverage can be obtained for various fault models.

The choice of LFSR seeds for compression methods is either random, e.g., as an outcome of Gauss-Jordan Elimination [21], or seed selection is designed for better seed compression [22]. These methods do not target the coverage of unmodeled faults. To enhance the effectiveness of LFSR reseeding for various defects and unmodeled faults, new techniques are needed for pattern modeling and seed selection.

We use the “output deviation” [23] as a surrogate coverage-metric for pattern grading in LFSR reseeding. A flexible, but general, probabilistic fault model is used to generate a probability map for the circuit, which can subsequently be used for LFSR seed selection. Seeds for the LFSR are selected for a precomputed set of test cubes to ensure that the resulting test patterns provide high output deviations. We show that such patterns provide high coverage for different fault models.

1.2.2 Seed Augmentation for LFSR Reseeding

In the deviation-based seed-selection method [24], seeds are carefully selected such that the patterns derived from them have high output deviation, which serves as a surrogate coverage metric. Test patterns with high output deviations have been shown to be effective for detecting unmodeled defects [24] [23]. However, a drawback of [24] is that each test cube is encoded by one seed, which results in a large number of seeds and high storage requirement. In [25] [26], methods have been proposed to reduce ATE storage by generating several child patterns from each parent pattern for modeled faults. However, their effectiveness for unmodeled defects has not been studied.

We examine the on-chip generation of additional seeds that are derived from a seed loaded from the tester. These seeds are generated by performing simple bit operations on the loaded seed. The proposed seed augmentation method is combined with the output deviations metric to ensure that, compared to [24], similar unmodeled

defect coverage is obtained with a smaller number of seeds. Here unmodeled defect coverage is represented by transition-delay and static bridging-fault coverage. We also show that when the same number of seeds are used as in [24], higher coverage ramp-up and higher unmodeled defect coverage are obtained for benchmark circuits. The bit operations are carried using simple on-chip hardware, and the area overhead and impact on test-application time are negligible.

1.2.3 Functional Test Grading at RT-Level

Very deep sub-micron process technologies are leading to increasing defect rates for ICs/boards [27] [28]. Since structural test alone is not enough to ensure high defect coverage, functional test is commonly used in industry to target defects that are not detected by structural test [29,30,83]. An advantage of functional test is that it avoids overtesting since it is performed in normal functional mode. In contrast, structural test is accompanied by some degree of yield loss [33]. RT-level fault modeling, DFT, test generation and test evaluation are therefore of considerable interest [34-37].

As described earlier, it is necessary to grade test sequences, reorder them, and select only a subset of the ordered sequences. The most effective test patterns are placed at the top of a reordered test sequence. In this way, the most effective test sets can be applied first during volume ramp-up, which increases the likelihood of detecting manufacturing defects in less time. The reason is that the defective chips will fail earlier with effective test sets applied first in the abort-at-first-fail system. Further, the test data volume will be greatly reduced since we can achieve high defect coverage using the highest-quality test sequences. The grading of tests is important since it can reduce both the test data volume and the test application time.

Given a large pool of functional test sequences (for example, test sequences used in design verification), it is necessary to develop an efficient method to select a subset of

sequences for manufacturing testing. Since functional test sequences are much longer than structural tests, it is time-consuming to grade functional test sequences using traditional gate-level fault simulation methods.

To quickly estimate the quality of functional tests, a high-level coverage metric for estimating the gate-level coverage of functional tests is proposed in [38]. This metric is based on event monitoring. First, gate-level fault activation and propagation conditions are translated to coverage objects at functional level. Next, a functional simulator is used for monitoring the “hit number” of the coverage objects. Finally, the fault coverage is estimated. However, this approach consumes an enormous amount of time and resources to extract the coverage objects. Also, it needs experienced engineers to extract efficient coverage objects.

In [39], another fault coverage metric is proposed to estimate the gate-level fault coverage of functional test sequences. This metric is based only on logic simulation of the gate level circuit, and on the set of states that the circuit traverses. Their work is based on the observation that a test sequence with a high fault coverage also traverses a large number of circuit states [40] [41]. The ratio of visited states to possible states for each subset is used to estimate the fault coverage. This method needs to perform gate-level logic simulation. It is infeasible when the gate-level netlist is not available and is impractical for large gate-level designs.

Although [38] and [39] propose methods to grade functional test sequences, their gradings are both performed at gate-level. Gate-level fault simulation is needed for the method proposed in [38] and gate-level logic simulation is needed for the method in [39]. An alternative solution is to grade functional test sequences at register-transfer (RT)-level based on some functional metric. In this thesis research, we have developed a deviation metric at RT-Level for this purpose.

The output deviations as a metric is defined at RT-level to grade functional test

sequences. The deviation metric at gate-level has been used in [23] to select effective test patterns from a large repository of n -detect test patterns. It has also been used in [24] to select appropriate LFSR seeds for LFSR-reseeding-based test compression. The deviation metric has been shown to be efficient at gate level. Here, we define the deviation metric at RT-level and use it for grading functional test sequences.

1.2.4 RTL DFT for Functional Test Sequences

A number of methods have been presented in the literature for test generation at RT-level. In [35], the authors proposed test generation based on a genetic algorithm (GA), targeting statement coverage as the quality metric. In [46–49], the authors used pre-computed test sets for RT-level modules (adders, shifters, etc.) to derive test vectors for the complete design. In [50], the authors presented a spectral method for generating tests using RT-level faults, which has the potential to detect almost the same number of faults as using gate-level test generation. In [51, 52], the authors proposed a fault-independent test generation method for state-observable finite state machines (FSMs) to increase the defect coverage.

To increase the testability of the complete design and to ease RT-level test generation, various DFT methods at RT-level have also been proposed. The most common methods are based on full-scan or partial scan. However, a scan-based DFT technique leads to long test application time and it is less useful for at-speed testing. On the other hand, non-scan DFT technique [53–58] offer low test application time and they facilitate at-speed testing. In [53], non-scan DFT techniques are proposed to increase the testability of RT-level designs. In [54], the authors presented a method called orthogonal scan. It uses functional datapath flow for test data, instead of traditional scan-path flow; therefore, it reduces test application time. In [55], a technique was proposed to improve the hierarchical testability of the data path, which can aid

hierarchical test generation. In [56], the authors presented a DFT technique for extracting functional control- and data-flow information from RT-level description and illustrated its use in design for hierarchical testability. This method has low overhead and it leads to shorter test generation time, up to 2-4 orders of magnitude less than traditional sequential test generation due to the use of symbolic test generation. In [57], the authors presented a method based on strong testability, which exploits the inherent characteristic of datapaths to guarantee the existence of test plans (sequences of control signals) for each hardware element in the datapath. Compared to the full-scan technique, this method can facilitate at-speed testing and reduce test application time. However, it introduces hardware and delay overhead. To reduce overhead, the authors proposed a linear-depth time-bounded testability-based DFT method in [58]. It ensures the existence of a linear-depth time expansion for any testable fault and experiments showed that it offers lower hardware overhead than the method in [57].

All the RT-level DFT methods described above attempt to increase the testability of the design to ease subsequent RT-level test generation. However, the functional test sequences for manufacturing test are derived in industrial practice from existing verification test sequences. These test sequences are generated by designers using manual or semi-automated means [59–61]. However, they often suffer from low defect coverage since they are mostly derived in practice from existing design-verification test sequences. Therefore, their effectiveness can be increased by using DFT techniques. Despite the large body of published work on RTL testing, prior work on RTL DFT has not been targeted towards increasing the defect coverage of existing functional test sequences.

We address the problem of improving the defect coverage of given functional test sequences for an RT-level design. The proposed method adopts the RT-level deviation

metric from [62] to select the most appropriate observation test points. While this approach can also be adapted for control-point insertion, we limit ourselves in this work to observation points. The deviation metric at the gate-level has been used in [23] to select effective test patterns from a large repository of n -detect test patterns. It has also been used in [63] to select appropriate LFSR seeds for reseeding-based test compression. The deviation metric at RT-level has been defined and used in [62] for grading functional test sequences.

1.2.5 Ranking Candidate Blocks for Diagnosis of Board-Level Functional Failures

Fault diagnosis is important in modern VLSI design and testing. It helps to find the root cause for failures in manufacturing testing.

It is common practice to use both structural and functional tests since they are complementary and cannot replace each other completely [29]. An advantage of functional test is that it can be used to evaluate power consumption, signal integrity, and noise-induced failures in native mode. In contrast, structural test is often accompanied by higher power consumption, di/dt problems, and yield loss. It may also lead to test escapes due to clock stretching [33].

A board may fail structural test or functional test, or both. Diagnosis techniques such as [13] [14], which have been developed to find the root cause for chip-level failures, cannot be used for fault diagnosis at board level when functional tests are applied. Moreover, external memory, noise, power-supply variations, and clock skew on a board aggravate the problem of board-level functional failure diagnosis [84].

The use of design-for-testability (DFT) techniques such as built-in self-test (BIST) and boundary scan can ease board-level diagnosis for structural test [84] [85]. However, in many cases in practice, a board does not fail until functional test is performed. In this case, the cost of root-cause identification is high since detailed failure analysis

is needed. In many situations, it is not even possible to find the root cause [86].

A common scenario in industry is “No Trouble Found” (NTF) due to functional failures. The problem here is that a component on a board fails board-level functional test, but it passes Automatic Test Equipment (ATE) test when it is returned to the component supplier for warranty replacement or service repair. Structural test diagnosis techniques are inadequate in such scenarios because it is difficult to produce errors due to NTFs in structural test mode.

Very little work has been published in the literature on techniques that can help to locate the root cause of a board-level functional failure. Trace buffers [87] [88] and the freezing/dump technique [89] [90] used in silicon debug are useful for localizing a board-level functional failure. However, trace buffers can provide real-time observation for only a few signals across a limited number of clock cycles. In the freezing/dump technique [89], the execution of normal functional is stopped, internal clocks are frozen, and values of scan cells and array contents are dumped. However, this method is not applicable to multiple clock domains driven by different PLLs, which is common in modern boards and systems. The freezing/dump feature based on a clock controller is also reported in [90]. However, the internal clock in normal functional mode is non-deterministic and non-repeatable, which makes it difficult to debug/diagnose functional failures.

In addition, due to design complexity, it is difficult to carry out fine-grained diagnosis for a board that fails functional test. A promising approach to address this problem is to carry out fault diagnosis in two phases—suspect faulty components on the board or modules within components (together referred to as blocks in this chapter) are first identified and ranked, and then more fine-grained diagnosis is carried out in the second phase to target the suspect blocks in ranked order.

In contrast to the second phase described above, relatively little attention has

been devoted to the first phase of diagnosis. In [86], a model-based reasoning technique is proposed for the diagnosis of board-level functional failures. A drawback of this approach is that it is hard to build a correct and complete model. In most cases, the success rate for diagnosis is considerably low during initial deployment. Additional effort is required to update the reasoning methods in the model to achieve higher success rates. In [91], two approaches based on Euclidean vectors and neural networks, respectively, are proposed to enhance the model-based reasoning engine. However, the computation time is impractical for both methods for large models. In [92], a board-level diagnosis framework based on Bayesian inference is proposed to infer the most likely faulty block on a failing board. However, considerable simulation time is needed to obtain the database of fault syndromes.

Take Figure 1.4 as an example. A component on a board can be partitioned into several modules according to their functionalities. We use the term *block* to refer to a component on a board as well as a major module within a component. We propose a new reasoning method based on dataflow analysis and Dempster-Shafer (DS) theory for ranking candidate blocks in the diagnosis of board-level functional failures. The candidate blocks are ranked such that the top-ranked blocks have the highest probability of being the root cause. The advantages of our method are as follows:

- In contrast to model-based reasoning, it does not require extensive effort and time to build a model;
- Time-consuming gate-level fault simulation for functional patterns is avoided. Instead, only logic simulation is needed;
- By targeting blocks in the ranked order, diagnosis time for the second phase can be significantly reduced;

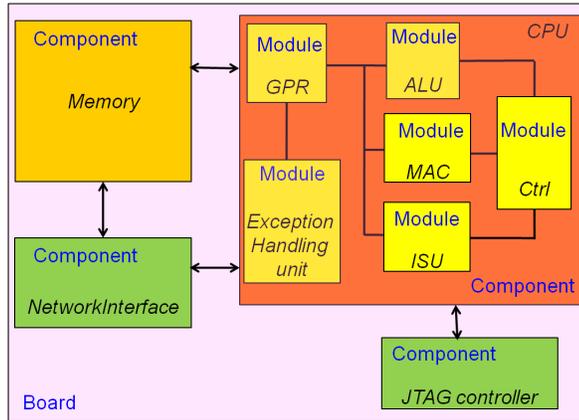


Figure 1.4: Blocks (components and modules within components) on a board.

- It does not rely on fault models. Therefore, accurate diagnosis can be expected for failures due to unmodeled defects.

1.2.6 Design-for-Testability and Diagnosis Methods for Board-Level Functional Failures

Although structural test suffers from the several disadvantages comparing with functional test, a major advantage of structural test is that it is applied in a controllable and deterministic manner, i.e., its clocks and signal values in each clock cycle is determined and repeatable in multiple runs with the same test. On the other hand, functional test is non-repeatable; the clock signal and signal values may show appreciable shifts in multiple runs for the same test. Therefore, it is hard to identify the exact time when a functional test fails in native (mission) mode and thus it is hard to diagnose functional failures.

Little work has been published in the literature on techniques that can help to locate the root cause of a board-level functional failure to a faulty wire/gate inside the component. This is necessary because additional test patterns can be generated once we have located the root cause to a faulty wire/gate. These additional test patterns can be ported to automatic-test-equipment (ATE) at the supplier side for

screening defective parts at an earlier stage, thus reducing defective-parts-per-million (DPPM) and product cost.

The debug and diagnosis of functional failures based on the trace buffer and freezing/dump techniques are useful for localizing a board-level functional failure. However, they all rely on a given functional test. Such an approach is labor-intensive due to the lack of automated and efficient diagnostic tools. Furthermore, even if the root cause has been found based on the given functional test, it is hard to recover and port the functional test into an ATE test program. To tackle these problems, a promising approach is to reproduce and detect functional failures using deterministic (i.e., repeatable) test. In this way, we can translate a board-level functional failure into a structural failure. The subsequent diagnosis procedure can be easily performed based on the structural failure because it is deterministic and repeatable. Furthermore, we can derive appropriate tests that can be ported to the ATE.

We propose a new technique, which we call *functional scan*, to bridge the gap between functional test and structural test. This technique can provide test conditions close to functional test in a controlled and repeatable manner. We also propose a new framework based on this technique to reproduce and detect the board-level functional failure using functional scan test. The proposed technique adopts the similar ideas of freezing and dumping of states. However, in prior work, internal clocks during normal functional mode are still non-controllable and non-deterministic when functional test is applied. In contrast, the proposed technique can provide controlled and deterministic internal clocks for both functional test and functional scan test. Furthermore, in the methods reported by Intel and Sun, the debug and diagnosis procedures are based on the given functional test. In our work, we first reproduce the functional failure using functional scan test and next debug/diagnose the failure detected by the functional scan test. The advantages of our proposed technique are:

1) we can reproduce and detect the functional failure in a controlled and deterministic environment; 2) we can diagnose the root cause to a faulty wire/gate inside a component; 3) we can provide the final functional scan test to the supplier, who can port it to the ATE.

Our goal is not to replace functional test or traditional structural test with functional scan test. Instead, we use functional scan test to reproduce and detect functional failures. This approach is different from pseudo-functional test [98], which has been proposed to address the yield loss problem in at-speed scan testing. The goal of pseudo-functional tests is to avoid functionally unreachable states. On the other hand, the purpose of functional scan test is to reproduce and detect functional failures in a deterministic environment when a board fails a functional test. Therefore, we refer to functional scan test as *deterministic test* in this thesis. Moreover, there are only two capture cycles in pseudo-functional tests while there are millions of capture cycles in functional scan test so that it can adequately mimic the test condition of the given functional test.

1.3 Thesis Outline

The remainder of this thesis is organized as follows.

In Chapter 2, we introduce the seed selection method for LFSR reseeding based test compression to target unmodeled defects. Experimental results for the ISCAS-89 and IWLS-05 benchmark circuits show that the selected seeds lead to high-deviation patterns, which provide higher defect coverage than patterns derived from other seeds. Moreover, faster fault-coverage ramp-up is obtained using these LFSR seeds. We have also shown that the patterns generated using the proposed method have negligible impact on test data volume. We have also highlighted the factors that influence the choice of p_1 and p_2 for the calculation of deviations. This choice is related to the

fraction of don't-care bits and the transition-fault coverage provided by the test cubes. In this Chapter, we also examine the on-chip generation of additional seeds that are derived from a seed loaded from the tester. These seeds are generated by performing simple bit operations on the loaded seed. The proposed seed-augmentation method is combined with the output deviations metric to ensure that, compared to [24], similar unmodeled defect coverage is obtained with a smaller number of seeds. We also show that when the same number of seeds are used as in [24], higher coverage ramp-up and higher unmodeled defect coverage are obtained for benchmark circuits. The bit operations are carried using simple on-chip hardware, and the area overhead and impact on test-application time are negligible.

In Chapter 3, we present the output deviation metric at RT-level to model the quality of functional test sequences. By adopting the deviation metric, timing-consuming fault simulation at gate-level can be avoided for the grading of functional test sequences. Experiments on the open-source Biquad filter core and an industry-strength Scheduler module show that the obtained deviations at RT-level correlate well with the stuck-at (transition/bridge) fault coverage at gate-level. Moreover, the reordered functional test sequences based on deviation method provide a steeper cumulative gate-level fault coverage curve. This is significant benefit in volume production. In this Chapter, we also present a modified RT-level deviations metric and shown how it can be used to select and insert the observation points for a given RT-level design and a functional test sequence. This DFT approach allows us to increase the effectiveness of functional test sequences (derived for pre-silicon validation) for manufacturing testing. Experiments on six *ITC'99* benchmark circuits show that the proposed RT-level DFT method outperforms two baseline methods for enhancing defect coverage. We also show that the RT-level deviations metric allows us to select a small set of the most effective observation points.

In Chapter 4, we propose a new method based on dataflow analysis and Dempster-Shafer theory for ranking faulty blocks in the diagnosis of board-level functional failures. The proposed approach transforms the information derived from one functional test failure into multiple-stage failures by partitioning the given functional test into multiple stages. A measure of “belief” is then assigned to each block based on the knowledge of each failing stage, and Dempster-Shafer theory is subsequently used to aggregate the beliefs from multiple failing stages. Blocks with higher beliefs are ranked at the top of the candidate list. Simulations on an industry design for a network interface application show that the proposed method can provide accurate ranking for most board-level functional failures. The proposed approach is computationally efficient since it avoids the need for time-consuming fault simulation procedures.

In Chapter 5, we propose an innovative functional test approach and DFT methods for the reproduction and detection of board-level functional failures. These DFT and test methods allow us to reproduce and detect functional failures in a controlled deterministic environment, which can provide ATE tests to the supplier for early screening of defective parts. Experiments on an industry design show that the proposed functional scan test with appropriate functional constraints can adequately mimic the functional state space, as measured by appropriate coverage metrics. Experiments also show that most functional failures due to stuck-at, dominant bridging, crosstalk and delay faults due to power supply noise can be reproduced and detected by functional scan test. We also describe two approaches to enhance the mimicking of the functional state space. The first approach allows us to select a given number of initial states in linear time and functional scan tests resulting from these selected states are used to mimic the functional state space. The second approach is based on controlled state injection.

Finally, Chapter 6 summarizes the contributions of the thesis and identifies directions for future work.

Chapter 2

Seed Selection and Seed Augmentation in Test Compression

LFSR reseeding forms the basis for many test compression solutions. A seed can be computed for each test cube by solving a system of linear equations based on the feedback polynomial of the LFSR. In this chapter, we focus on the seed selection and seed augmentation in test compression. We first introduce the probabilistic fault model and the theory of output deviations. Next we present the proposed seed selection method based on output deviations. In this approach, output deviation measure is used as a surrogate coverage-metric to select appropriate LFSR seeds. We also present the seed augmentation method to further reduce the requirement of memory usage.

2.1 Probabilistic Fault Model and Output Deviations

In this section, we review the general fault model used in this work and the concept of output deviations. The *confidence level* (CL) of a single-output gate encompasses all the different input combinations of the gate, and for a given input combination, it provides the probability that the gate output is correct for the corresponding input combination. The probability that a gate output is correct can be different for the various input combinations [23].

The confidence level R_i of a gate G_i with m inputs and a single output is a vector with 2^m components, defined as: $R_i = (r_i^{(00\dots00)} r_i^{(00\dots01)} r_i^{(00\dots10)} \dots r_i^{(11\dots11)})$, where each component of R_i denotes the probability that the gate output is correct for the

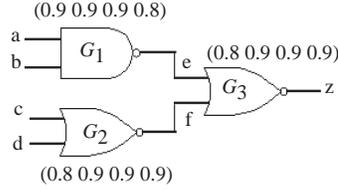


Figure 2.1: An example to illustrate confidence levels.

Table 2.1: Signal probabilities for different input combinations.

Input pattern, z	$p_{e,0}$	$p_{e,1}$	$p_{f,0}$	$p_{f,1}$	$p_{z,0}$	$p_{z,1}$
0000, 0	0.1	0.9	0.2	0.8	0.886	0.114
0101, 0	0.1	0.9	0.9	0.1	0.837	0.163
1111, 1	0.8	0.2	0.9	0.1	0.396	0.604

corresponding input combination.

We associate signal probabilities $p_{i,0}$ and $p_{i,1}$ with each line i , where $p_{i,0}$ and $p_{i,1}$ are the probabilities for line i to be at logic 0 and 1, respectively. To reduce the amount of computation, signal correlations due to reconvergent fanout are not considered.

Let i be the output of a two-input gate G , and let j and k be its input lines. If G is a NAND gate, we have:

$$p_{i,0} = p_{j,1}p_{k,1}r_i^{(11)} + p_{j,0}p_{k,0}(1-r_i^{(00)}) + p_{j,0}p_{k,1}(1-r_i^{(01)}) + p_{j,1}p_{k,0}(1-r_i^{(10)})$$

$$p_{i,1} = p_{j,0}p_{k,0}r_i^{(00)} + p_{j,0}p_{k,1}r_i^{(01)} + p_{j,1}p_{k,0}r_i^{(10)} + p_{j,1}p_{k,1}(1-r_i^{(11)})$$

Likewise, the signal probabilities can be easily computed for other gate types. Fig. 2.1 shows a circuit consisting of three gates G_1 , G_2 , and G_3 with different CLs. For this circuit, Table 2.1 lists the signal probabilities for three different input vectors. The fault-free values at the output z are also listed in the first column of Table 2.1.

For any logic gate (or primary output) g in a circuit, let its fault-free output value for any given input pattern t_j be d , $d \in \{0, 1\}$. The *output deviation* $\Delta_{g,j}$ of g for input pattern t_j is defined as $p_{g,\bar{d}}$, where \bar{d} is the complement of d . Intuitively, the deviation for an input pattern is a measure of the likelihood that the gate output

is incorrect for that input pattern. The output deviations for the three patterns in Table 2.1 are highlighted. Output deviations can be determined without explicit fault grading, hence the computation is feasible for large circuits and large test sets.

We use the following arbitrarily chosen set of CLs for our experiments. It has been shown in prior work [23] that the pattern-grading results are relative insensitive to small variations in the values of CL. These vectors are defined separately for each gate type. For example, for a 2-input NAND gate, we use (1) low CL: $R^{NAND2} = (0.8^{(00)}, 0.8^{(01)}, 0.8^{(10)}, 0.7^{(11)})$, and (2) high CL: $R^{NAND2} = (0.95^{(00)}, 0.95^{(01)}, 0.95^{(10)}, 0.85^{(11)})$.

These CLs are chosen to reflect the fact that, when both inputs are non-controlling, the probability for the gate to produce the correct output is lower than for other input combinations. Since both sets of CLs yield similar results, we only report results obtained using the “high CL” values.

The output deviation metric was used in [64] to select effective test patterns from a large repository n -detect test set. Test selection is important for time-constrained and wafer-sort environments that have strict budgets on test data volume and test time. If highly effective test patterns are applied first, defective chips will fail earlier, reducing test application time in an abort-at-first-fail environment. Experimental results in [64] show that for the same test length, test patterns selected using output deviations are consistently more effective than patterns selected using other methods, in terms of the coverage for resistive shorts, wired-AND/OR bridging faults, and gate exhaustive metrics [65].

2.2 LFSR Reseeding Targeting Unmodeled Defects

Despite the availability of numerous LFSR-reseeding-based compression methods in the literature, relatively little is known about the effectiveness of these seeds for

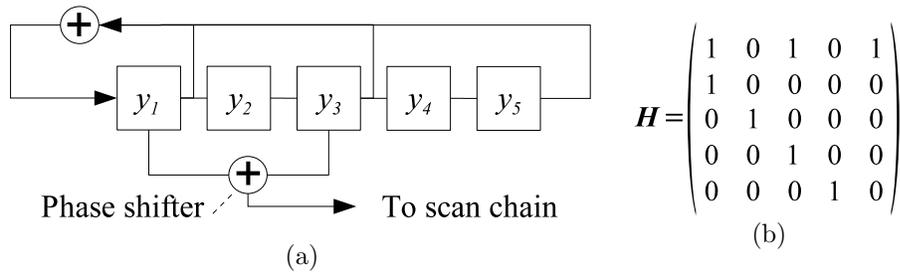


Figure 2.2: (a) An LFSR and phase shifter; (b) State transition matrix of the LFSR.

unmodeled defects. In this section, We have used the recently proposed output deviation measure of the resulting patterns as a metric to select appropriate LFSR seeds [63]. Experimental results are reported using test patterns for stuck-at and transition faults derived from selected seeds for ISCAS-89 and IWLS-05 benchmark circuits. These patterns achieve higher coverage for transition and stuck-open faults than patterns obtained using other seed-generation methods for LFSR reseeding. Given a pattern pair (p_1, p_2) for transition faults, we also examine the transition-fault coverage for launch-on-capture by using p_1 and p_2 to compute output deviations. Results show that p_1 tends to be better when there is a high proportion of don't-care bits in the test cubes, while p_2 is a more appropriate choice when the transition fault coverage is high.

2.2.1 Seed Selection

Given the scan-chain architecture for the circuit under test, LFSR feedback polynomial, and phase shifter (if any), the operation of the LFSR and phase shifter can be symbolically simulated to determine a system of linear equations for each test cube. The resulting system of linear equations has the form $\mathbf{A}\vec{y} = \vec{z}$, where \mathbf{A} is a matrix that can be derived from the LFSR feedback polynomial and the phase shifter, \vec{z} is a column vector corresponding to the specified bits in the test cube, and the solution for the vector \vec{y} is the seed.

Fig. 2.2 shows an external-XOR LFSR [2] with the feedback polynomial $x^5 + x^3 + x + 1$, and a 1-stage phase shifter. The state of the LFSR can be represented using a vector $\vec{S} = (s_1, s_2, \dots, s_N)^t$, where N is the size of the LFSR and s_1 (s_N) corresponds to the leftmost (rightmost) stage. The j -th state of the LFSR is derived recursively as $\vec{S}_j = \mathbf{H}\vec{S}_{j-1}$, $j = 1, 2, \dots$, where \mathbf{H} is the state transition matrix for the LFSR.

The j -th output of the 1-stage phase shifter shown in Fig. 2.2 can be represented as $O_j = \vec{P}^t \vec{S}_j = \vec{P}^t \mathbf{H}^j \vec{S}_0$, $j = 1, 2, \dots$. Vector \vec{P} represents the operation of the phase shifter. If stage j of the LFSR is connected to the XOR gate, the j -th row in \vec{P} is set to ‘1’. For the phase shifter in Fig. 2.2, we have $\vec{P} = (10100)^t$. For example, the second output of the phase shifter is $O_1 = \vec{P}^t \mathbf{H} \vec{S}_0 = (11101) \vec{S}_0 = y_1 + y_2 + y_3 + y_5$.

For the test cube 101xxxxx (the leftmost bit ‘1’ is loaded into the first scan cell that is next to the scan out pin), we can obtain a system of linear equations as shown in Fig. 2.3(a). Gauss-Jordan Elimination [21] can be used to transform a set of columns in \mathbf{A} into an identity matrix (these columns are referred to as *pivots*) while the remaining columns are *free-variables*, as shown in Fig. 2.3(b). The set of solutions for the pivots can be represented as a linear combination of the free-variables, as shown in Fig. 2.3(c). To obtain multiple seeds for each test cube, we first make random assignments to free-variables and then compute pivots.

The high-level flow of LFSR seed selection is shown in Fig. 2.4(a). Given a set of test cubes T , we generate N_s seeds for each test cube (Lines 2-6). Each seed is expanded into a fully-specified test pattern (Line 4). All seeds and their corresponding patterns are stored in two sets, S and V respectively. In Line 8, test patterns in V are sorted and a new sorted test set V' is obtained. To obtain a seed set, in Lines 9-13, we sequentially select the top most pattern in V' and its corresponding seed. Once a test pattern is selected (Line 10), all other test patterns that are derived from the same test cube in Line 4 are removed from V' (Line 12). Procedure 1 in Fig. 2.4(a) is

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$$

(a) System of linear equations

$$\left(\begin{array}{cc|c} \text{Pivots} & \text{Free Variables} & \\ \hline 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{array} \right)$$

(b) Gauss-Jordan elimination

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = y_4 \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} + y_5 \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$

(c) Solution space

Figure 2.3: Example to illustrate the solution for a system of linear equations.

generic. A different sort method in Procedure Sort_Pattern leads to a different seed set.

In this Chapter, we use output deviation as a metric for seed selection, i.e., test patterns in V are sorted according to their output deviations, such that test patterns with high deviations can be selected earlier than test patterns with low deviations. As shown in Fig. 2.4(b), for each primary output (PO), all test patterns in T are sorted in descending order based on their output deviations. The result is stored in a matrix X with M columns and $|T|$ rows, where M is the number of POs (Line 1). The element in the i -th row and the j -th column of X is the test pattern that has the i -th highest output deviation at the j -th PO. Fig. 2.5 illustrates this sorting procedure. From matrix X , a new ordered test set T' can be obtained. For the example in Fig. 2.5, the resulting sorted test set is $T' = \{t_1, t_3, t_5, t_2, t_4, t_6\}$.

To evaluate the effectiveness of a set of seeds for defect screening and for getting steeper fault coverage ramp-up, we consider the fault coverage obtained using the test patterns that are derived from the seeds for different fault models, including stuck-at

Inputs: T ,
 N_s

- 1: **for all** test cube t_p ,
 $p = 1, 2, \dots, |T|$ **do**
- 2: **for** $i = 1$ **to** N_s **do**
- 3: generate a new seed $s_{p,i}$;
- 4: expand seed $s_{p,i}$ into a test pattern $v_{p,i}$;
- 5: add $v_{p,i}$, $s_{p,i}$ to sets V , S ;
- 6: **end for**
- 7: **end for**
- 8: $V' = \text{Sort_Pattern}(V)$;
- 9: **while** $V' \neq \emptyset$ **do**
- 10: select the top-most pattern, namely $t_{p,k}$;
- 11: select seed $s_{p,k}$;
- 12: remove patterns $t_{p,i}$ from V' , $i = 1, 2, \dots, N_s$;
- 13: **end while**

(a) Procedure 1

Inputs: T -test set
Output: T' -sorted test set

- 1: sort test patterns by deviations at each PO, obtain X ;
- 2: **while** $|T'| < |T|$ **do**
- 3: **for** $j = 1$ **to** M **do**
- 4: **for** $i = 1$ **to** $|T|$ **do**
- 5: **if** $X[i][j] \notin T'$ **then**
- 6: add $X[i][j]$ to T' ;
- 7: **break**;
- 8: **end if**
- 9: **end for**
- 10: **end for**
- 11: **end while**

(b) Procedure 2

Figure 2.4: Procedures: (a) LFSR reseeding with seed selection, (b) Sort test patterns according to output deviations.

Pattern ID	Output deviations		
	Port A	Port B	Port C
t_1	0.9	0.2	0.7
t_2	0.8	0.3	0.4
t_3	0.6	0.7	0.8
t_4	0.2	0.6	0.5
t_5	0.7	0.4	0.9
t_6	0.3	0.1	0.2

(a) Test patterns and output deviations.

Port A	Port B	Port C
t_1	t_3	t_5
t_2	t_4	t_3
t_5	t_5	t_1
t_3	t_2	t_4
t_6	t_1	t_2
t_4	t_6	t_6

(b) Outcome of the sorting procedure (Matrix X).

Figure 2.5: An example for the sorting of test patterns based on output deviations.

faults, transition faults, and stuck-open faults. To detect sequence-dependent stuck-open and transition faults, we use the launch-on-shift (LOS) and launch-on-capture (LOC) schemes to apply a test pattern pair (p_1, p_2) to the CUT. These methods are also referred to as skewed-load and broadside testing [2], respectively. Fig. 2.6 provides an example. Given a test cube from an n -detect stuck-at test set or a 1-detect LOC transition-fault test set, a seed is computed and a "high-deviation" test pattern p_0 is derived from the seed.

We first consider the case when p_0 is derived from a stuck-at test cube. In the LOS scheme, p_2 is obtained by shifting p_1 one clock cycle. Hence, as shown in Fig. 2.6(b), p_1 is obtained by shifting the first $L - 1$ bits of p_0 (L is the length of the scan chain), and p_2 is identical to p_0 . In the LOC scheme, as shown in Fig. 2.6(c), p_2 is obtained by capturing the responses of p_1 , and p_1 is identical to p_0 . For both the LOS and LOC schemes, output deviations are first computed directly using the test patterns in V (Fig. 2.4(a), Line 8). The resulting seed set is referred to as S_{dev1} . For the LOC scheme, we also generate another seed set in the following manner: First obtain the responses of the patterns in V , and then compute output deviations using these responses. The resulting seed set is referred to as S_{dev2} . We also assume that the combinational primary inputs remain unchanged for p_1 and p_2 .

Next, we consider the case when p_0 is derived from a LOC transition-fault test cube. In this case, p_2 is obtained by capturing the responses of p_1 . We use p_2 to calculate deviations and the resulting seed set is referred to as S_{dev3} .

Table 2.2 shows the various seed sets obtained using pattern pairs (p_1, p_2) . For the case of stuck-at test cubes, only S_{dev1} is generated in the LOS test application method. For the LOC test application method, both S_{dev1} and S_{dev2} are generated. We obtain S_{dev1} when p_1 is used to generate seeds and obtain S_{dev2} when p_2 is used. For the case of transition-fault test cubes, only S_{dev3} is obtained.

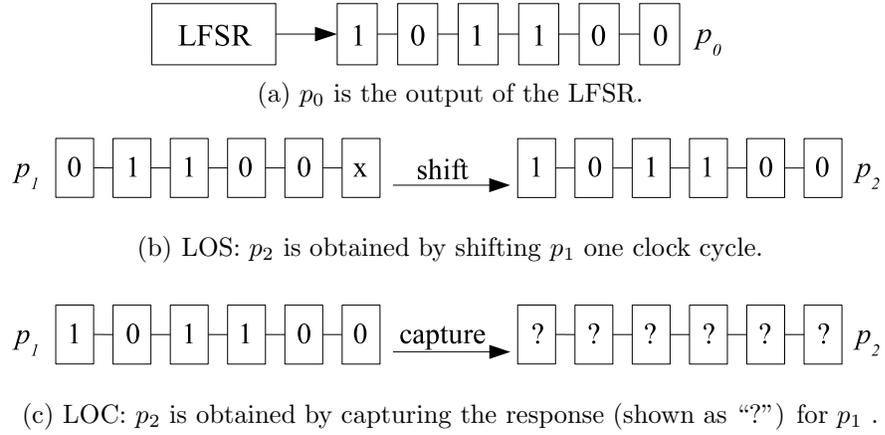


Figure 2.6: Applying a test pattern pair using LOS and LOC schemes.

Table 2.2: The various seed sets obtained using pattern pairs (p_1, p_2) .

Fault model for precomputed test cubes	Test application method	Pattern used to generate seeds	Seed set
Stuck-at	LOS	p_2	S_{dev1}
	LOC	p_1	S_{dev1}
		p_2	S_{dev2}
Transition	LOC	p_2	S_{dev3}

For comparison purposes, we also generate three other seed sets, referred to as S_{rand} , S_{greedy} , and S_{comp} , respectively. The seed set S_{rand} is obtained by randomly generating one seed for each test cube. The seed set S_{greedy} is generated by sorting test patterns in V by the number of hard stuck-at faults that they detect. Hard faults are ones that are not detected by a predefined number (we use 128, 256, or 1024 for the different benchmarks in our experiments) of pseudo-random patterns. The detection count is obtained using fault simulation without fault dropping.

The seed set S_{comp} is used as a baseline to evaluate the impact on storage requirement for the deviation-based seed selection method. In [22], a Huffman-encoding-based compression method is proposed. The LFSR is divided into multiple blocks, with each block having k stages (the last block may have less stages if the k cannot divide the LFSR size N). Consequently, each N -bit LFSR seed is divided into multiple k -bit segments. Each segment can be viewed as a k -bit symbol and these symbols are compressed using Huffman encoding. To improve the encoding efficiency, when generating seeds for S_{comp} , the system of linear equations is solved to ensure that the most significant bits of the symbols are biased toward 0. Hence, seeds in S_{comp} have more “0” bits than seeds in other seed sets, which may result in lower defect coverage. To evaluate the impact of the seed-selection method on the effectiveness of seed compression, we used the method from [22] to compress S_{dev1} , S_{dev2} , S_{dev3} , and S_{comp} .

2.2.2 Experimental Evaluation

To evaluate the effectiveness of our seed-selection procedures, experiments are carried out on the ISCAS-89 and the IWLS-05 benchmark circuits; the latter set contains larger circuits than the ISCAS-89 benchmarks. The experiments for the ISCAS-89 circuits are carried out using the n -detect stuck-at test cubes, while the experiments

for the IWLS-05 benchmarks are carried out using both n -detect stuck-at and 1-detect transition-fault test cubes.

Results for ISCAS-89 Circuits

Experimental Setup

All experiments for ISCAS-89 were performed on 64-bit Linux servers with 4GB memory each. The n -detect stuck-at test cubes were generated using proprietary tools from industry; we also used these tools to carry out fault simulation for various fault models. The deviation calculation, seed selection, and other related programs were coded in C++.

To obtain the transition-fault coverage, we used LOC fault simulators. The LOC transition fault simulator assumes two constraints: (1) primary inputs remain unchanged during the two capture cycles, and (2) primary outputs are not probed to detect faults. The reason is that for at-speed testing, low-speed testers usually cannot drive or probe pins at the functional clock frequency, which is much faster than the scan-test frequency.

Results for 5-detect Stuck-at Test Cubes

For each ISCAS-89 circuit, we first obtain an n -detect set of test cubes as described in Figure 1 of [66]. During the ATPG process, after a new set of test cubes is generated, the ATPG tool randomly fills don't-cares in these test cubes and performs fault simulation to drop faults that are fortuitously detected by these patterns and have been detected n times. Next, a procedure called *Xize* is invoked to identify as many bits as possible that can be relaxed to don't-cares without any loss of n -detect fault coverage. Due to space limits, we only report results obtained with $n = 5$.

From the n -detect test cubes, the seed sets described in Section 2.2.1 are generated. For each circuit, four cases are considered corresponding to different fault

models and test-application schemes:

- stuck-at faults;
- stuck-open faults detected using the LOS scheme (the industry tool reports fault coverage for stuck-open faults only for LOS);
- transition faults detected using the LOC scheme. We do not consider LOS here because LOC is more practical and commonly used in industry.

The fault coverage for these fault models are obtained using patterns derived from the different seed sets.

Fig. 2.7-2.8 show the fault coverage obtained for various test lengths for different fault models for several ISCAS-89 circuits. For stuck-open and transition faults, the deviation-based approach yields the best results. The seeds selected using the proposed approach provide higher defect coverage with a smaller number of patterns. For stuck-at faults, the results for the various methods are comparable. This is expected because the precomputed test sets are tailored for stuck-at coverage.

It can be seen from Case 2 and Case 3 that, as the number of test patterns applied to the CUT increases, the difference in the fault coverage for the different seed sets decreases, i.e., the fault coverage tends to converge to a maximum level. This is because, by first applying test patterns derived from effective LFSR seeds, we can cover most easy-to-detect stuck-open and transition faults using the first few test patterns (40%-50% for most benchmark circuits). In Case 3, the fault coverage for the complete set of patterns is relatively low, because the second test pattern p_2 is not directly generated by the n -detect ATPG tool. These observations suggest that, in order to obtain high defect coverage with a small number of test patterns, we should only apply a subset of the n -detect test patterns (with seeds determined using

Table 2.3: Comparison of compression results.

Circuit	Seed Size (bits)	Block Size	S_{dev1}, S_{dev2}		S_{comp}	
			Vol.	Comp%	Vol.	Comp%
s5378	158	32	37504	58.73	31391	65.45
s9234	158	32	58938	61.10	49011	67.65
s13207	247	64	62592	79.95	35272	88.70
s15850	274	64	44134	78.29	34917	82.82
s38417	661	64	96220	78.52	94994	78.80
s38584	520	64	94624	77.69	83293	80.37

output deviations) and then add top-off ATPG patterns targeting the remaining hard-to-detect faults. The high coverage provided by a small number of high-deviation patterns (for different fault models) leads to a reduction in the test data volume that is required to predict high defect coverage.

Finally, we compare the storage requirement for S_{dev1} (S_{dev2}) and S_{comp} , as shown in Table 2.3. The compression results for S_{dev2} and S_{dev1} are identical for all the benchmark circuits. In Table 2.3, columns named “Vol.” denote the resulting seed volume in bits after compression, and columns named “Comp%” indicate the corresponding compression percentage relative to the set of seeds. As can be seen, the test data volume for the proposed deviation-based seed selection technique is only slightly higher than the test data volume for [22], while achieving significantly higher defect coverage. It is also noteworthy that the seed volume reported in Table 2.3 only includes the seed data stored on the tester. For large industrial designs, the area overhead of the on-chip Huffman decoder may be prohibitively large, rendering the compression scheme of [22] infeasible.

Selection of p_1 or p_2 in Calculation of Output Deviations

For the six ISCAS-89 circuits, we have separately evaluated the LOC transition-fault coverage for patterns derived from S_{dev1} and S_{dev2} . (Recall that S_{dev1} is the seed-set selected using the deviations for p_1 , and S_{dev2} is the seed-set selected using the deviations for p_2 in the LOC scheme.) The following discussion explains how an appropriate choice (p_1 or p_2) can be made based on the properties of the test set.

Table 2.4: Selection of p_1 or p_2 in calculation of deviations.

Circuit	Flow1 (Proprietary ATPG tool)		Flow2 (Tetramax)		
	$x\%$	p_1 or p_2 ?	$fc\%$	$x\%$	p_1 or p_2 ?
s5378	72.56	p_2	75.67	96.07	p_2
s9234	74.60	p_2	32.64	95.38	p_1
s13207	92.87	p_2	60.96	98.91	p_2
s15850	86.93	p_2	26.57	98.81	p_1
s38417	82.28	p_2	84.53	99.33	p_1
s38584	86.55	p_2	16.70	99.50	p_1

We carried out a set of experiments using 1-detect stuck-at test cubes. Experiments were performed using two flows. In the first flow, test cubes were generated using a proprietary tool from industry. In the second flow, test cubes were generated using the Tetramax tool from Synopsys. Table 2.4 compares the results obtained using these two flows. The second and fifth columns ($x\%$) list the average percentage of don't-care bits in a test cube. The third and sixth column (p_1/p_2) indicates which seed-generation method (based on p_1 or p_2) yields higher transition-fault coverage. The fourth column ($fc\%$) shows the transition-fault coverage provided by the patterns derived from S_{dev2} in the second flow.

From Table 2.4, we can see that two factors ($x\%$ and $fc\%$) determine whether p_1 or p_2 should be used to calculate deviations. Among these two factors, $x\%$ appears to be more important. We make the following key observations:

- When $x\%$ is low (e.g., $x\% < 93\%$ for these six benchmarks), the seeds in S_{dev2} are better than those in S_{dev1} . In other words, p_2 should be used for calculating deviations.
- When $x\%$ is high (e.g., $x\% > 99\%$ for these six benchmarks), the seeds in S_{dev1} are better than those in S_{dev2} . In other words, p_1 should be used for calculating deviations.

- When $x\%$ is moderately high (e.g., $93\% < x\% < 99\%$ in Table 2.4) and $fc\%$ is low ($fc\% < 33\%$ in Table 2.4), it is more appropriate to compute deviations using p_1 .
- When $x\%$ is moderately high (e.g., $93\% < x\% < 99\%$ in Table 2.4) and $fc\%$ is high ($fc\% > 60\%$ in Table 2.4), p_2 should be used for computing deviations.

A low value of $x\%$ implies that seed selection based on p_1 has less degree of freedom, which adversely affects the quality of the resulting patterns. On the other hand, a large value of $x\%$ implies that seed selection using p_1 has more degree of freedom, which leads to high-quality seeds. Where $x\%$ is small (e.g., $x\% < 93\%$ in Table 2.4) or very high (e.g., $x\% > 99\%$ in Table 2.4), it plays a dominant role in determining whether p_1 or p_2 is more efficient for deviation calculation. The impact of $fc\%$ can be ignored in these cases. For example, in the first flow, the use of p_2 consistently leads to better results than the use of p_1 because the value of $x\%$ is rather low ($x\% < 93\%$). However, for the second flow, for $s38417$ and $s38584$, p_1 is preferred to p_2 since the value of $x\%$ is very high ($x\% > 99\%$) for both of them.

When the value of $x\%$ is neither very large or very small (e.g., between 93% and 99% in Table 2.4), $fc\%$ must also be considered. A relatively high value of $fc\%$ ($fc\% > 60\%$ here) implies that it is easy to activate the transition faults. When the transition faults can be easily activated, p_2 becomes more important since it is used to capture the fault effects; hence p_2 should be used to calculate deviations. In the second flow, p_2 is better than p_1 for $s5378$ and $s13207$ since the following conditions are satisfied: the value of $x\%$ is between 93% and 99% , and the value of $fc\%$ is more than 60% . On the other hand, low $fc\%$ ($fc\% < 33\%$ in Table 2.4) indicates that it is difficult to activate the transition faults. In this case, p_1 becomes more important (e.g., as shown by the results for $s9234$ and $s15850$ in the second flow). The above qualitative discussion can serve as a guideline for deviation calculation.

Results for IWLS-05 Benchmarks

In order to evaluate the effectiveness of the proposed seed-selection method for larger circuits, we carried out experiments using the IWLS-05 benchmarks. We first briefly describe the IWLS-05 benchmarks and we then present results for n -detect ($n = 1, 3$) stuck-at test cubes and 1-detect LOC transition-fault test cubes. We consider transition-fault test cubes because it is common practice in industry to first apply transition-fault patterns, then apply additional top-off stuck-at patterns during manufacturing testing.

IWLS-05 Benchmarks

We used 8 designs from the recently-released set of IWLS-05 benchmarks. Table 2.5 shows the characteristics of the selected benchmarks, including their sizes, and the number of stuck-at faults.

Table 2.5: Characteristics of IWLS-05 benchmarks.

Benchmark name	No. of gates	No. of scan cells	No. of stuck-at faults
tv80	13978	404	73562
mem_ctrl	25380	1350	124686
ac97_ctrl	33004	2302	146136
DMA	49960	3131	230338
pci_bridge32	52163	3720	231050
wb_conmax	73783	3316	366720
des_perf	167592	9105	773570
ethernet	176075	10752	811084

Experimental Setup

All experiments for the IWLS-05 benchmarks were performed on 64-bit Linux servers with 4 GB memory each. The n -detect stuck-at and 1-detect LOC transition test cubes were generated using the Tetramax tool from Synopsys, which was also used to run fault simulations for different fault models. The Synopsys Verilog Compiler (VCS) was used to run Verilog simulations and compute the output deviations

for benchmarks. All other programs were implemented in C++.

As before, we obtained the transition-fault coverage under the LOC test-application method. The constraints used in the LOC scheme are the same as that for the ISCAS-89 circuits.

Experiments Based on n -detect Stuck-at Test Cubes

For each benchmark, we first obtained the n -detect stuck-at test cubes using the Tetramax tool from Synopsys. Table 2.6 presents the details of the n -detect ($n = 1, 3$) test cubes that we generated. If the number of stuck-at test cubes is more than 50,000 for a benchmark, we only use the first 50,000 test cubes. We do this because of two reasons:

1. For various seed generation methods, there is only a slight difference in cumulative coverage after 50,000 patterns, thus it is sufficient to select only the first 50,000 seeds to evaluate pattern quality.
2. For each benchmark, due to limited memory, several rounds are needed to complete the seed selection procedure. In each round, we sort $K \cdot N_s$ (recall that N_s is the number of seeds generated for each test cube) patterns and obtain K seeds that lead to K high-deviation patterns. In our experiments, K is set to be less than 10,000. Therefore, the 50,000 seeds selected for the 50,000 test cubes are nearly identical to the first 50,000 seeds obtained if the complete set of test cubes is used.

The seed sets described in Section 2.2.1 are next generated from the n -detect test cubes. For each benchmark, we report the cumulative fault coverage obtained for transition faults using the LOC method. The stuck-at fault coverage is nearly identical for all the seed sets, therefore it is not shown here. The choice of either p_1 or p_2 for deviation calculation is based on the arguments presented in Section 2.2.2.

Table 2.6: Characteristics of n -detect stuck-at test cubes.

Benchmark name	1-detect		3-detect	
	No. of test cubes	Fault coverage	No. of test cubes	Fault coverage
tv80	7008	97.36	24800	97.32
mem_ctrl	10649	79.24	32768	79.24
ac97_ctrl	20146	99.66	60268	99.66
DMA	26782	84.14	89408	84.12
pci_bridge32	31312	99.38	93982	99.38
wb_conmax	57681	95.12	173173	95.12
des_perf	76001	99.83	226532	99.83
ethernet	119636	99.85	358194	99.85

The cumulative fault coverage for each case is obtained using the patterns derived from the different seed sets.

Fig. 2.9-2.14 show the fault coverage obtained for different fault models. Fig. 2.9-2.11 are based on the 1-detect stuck-at test cubes, and Fig. 2.12-2.14 are based on the 3-detect stuck-at test cubes. For the three largest benchmarks `wb_conmax`, `des_perf`, and `ethernet`, we first dropped the easy-to-detect faults by using either 256 or 10,000 pseudorandom patterns, and then generated 1-detect stuck-at test cubes for the remaining faults. The result for `ethernet` is shown in Fig. 2.11(b). The results for `wb_conmax` and `des_perf` are not shown since we obtained similar LOC transition fault coverage for them by using different seed generation methods. For LOC-based testing for transition faults, the deviation-based approach clearly outperforms the other methods for most circuits. Moreover, the seeds selected using the proposed approach provide higher coverage with a smaller number of patterns.

Another observation is that the first few test patterns can cover most easy-to-detect transition faults if these test patterns are derived from effective LFSR seeds, which implies that in order to obtain high defect coverage with a small number of test patterns, we should only apply a subset of the n -detect test patterns (with seeds determined by output deviations) and then add top-off ATPG patterns targeting

the remaining hard-to-detect faults. The high coverage provided by a small number of high-deviation patterns (for different fault models) reduces the test-data volume required to achieve high defect coverage.

Experiments Based on 1-detect LOC Transition Test Cubes

Since it is common practice in industry to apply transition-fault test patterns using LOC, we also carried out experiments on 1-detect LOC transition-fault test cubes. Table 2.7 shows the fault coverage obtained using transition-fault test patterns.

Table 2.7: 1-detect LOC transition-fault test cubes.

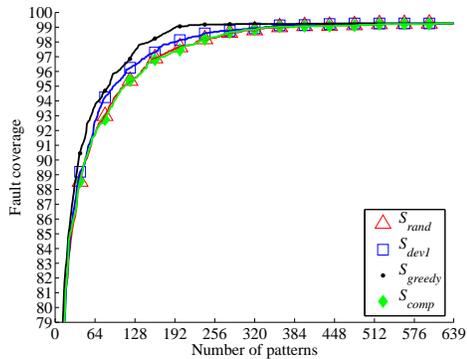
Benchmark name	No. of faults	No. of 1-detect transition-fault test cubes	Fault coverage
tv80	71942	18387	96.35
mem_ctrl	119274	25258	68.09
ac97_ctrl	136924	38127	99.61
DMA	217806	50593	80.30
pci_bridge32	216166	61962	99.38
wb_conmax	353452	116549	95.02
des_perf	737146	186100	99.85
ethernet	768072	251728	99.73

As in the case of stuck-at test cubes, seeds are generated for the 1-detect LOC transition-fault test cubes, and subsequently expanded to test patterns. For the LOC transition-fault test cubes, both p_1 and p_2 are generated using TetraMax. Since p_2 is directly generated by TetraMax, we use p_2 to calculate the deviation for each pattern. Seeds are selected based on the obtained deviation data, with the selected seed set denoted by S_{dev3} (Table 2.2). Test patterns are derived from the four seed sets obtained from different methods. For these pattern sets, we compare the cumulative LOC transition fault coverage and stuck-at coverage.

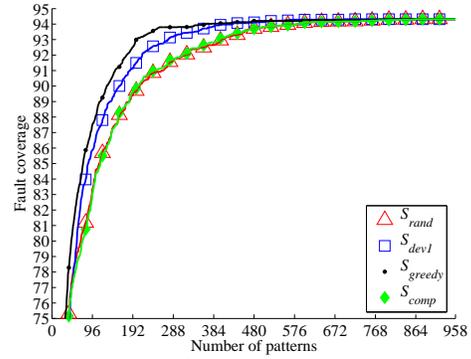
Fig. 2.15-2.16 compare the cumulative stuck-at and transition fault coverage (LOC scheme) for the different seed sets. We only show the cumulative coverage of

the first 5000 or 10,000 patterns since the coverage difference is not obvious for the different seed sets after 5000 or 10,000 patterns. We can see that for most of the benchmarks, S_{dev3} provides the best fault coverage among all the seed sets, for both stuck-at fault model and transition fault model. This demonstrates the effectiveness of the proposed seed selection method for transition test cubes.

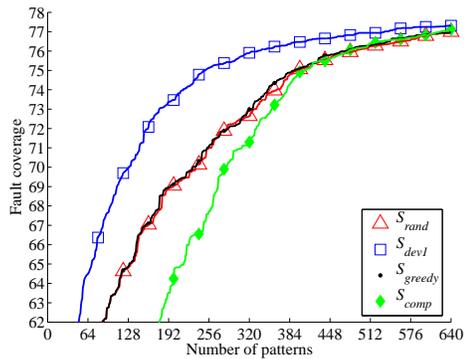
Finally, we note that for the IWLS-05 benchmark circuits, the seeds obtained using [22] can be compressed more effectively than the seeds obtained using the proposed method. The seeds derived using [22] have very long runs of 0s. Nevertheless, the difference is not particularly significant because the length of the seeds for these circuits is already much smaller (43X on average) than the test-pattern lengths.



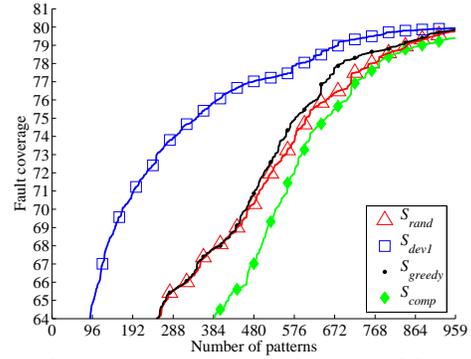
Case (1): stuck-at faults



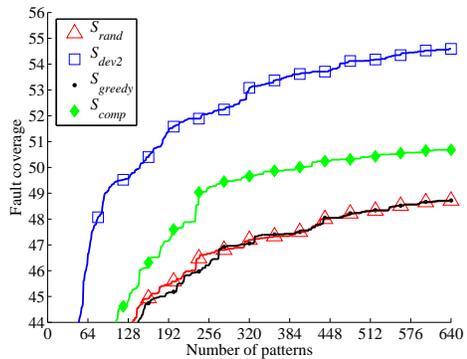
Case (1): stuck-at faults



Case (2): stuck-open faults (LOS)

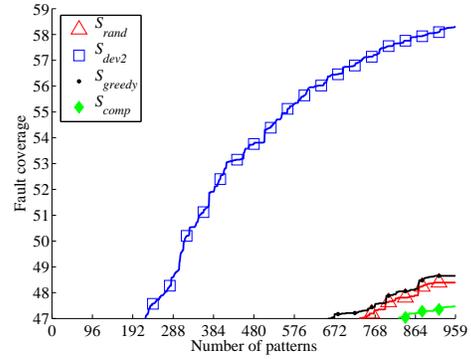


Case (2): stuck-open faults (LOS)



Case (3): transition faults (LOC)

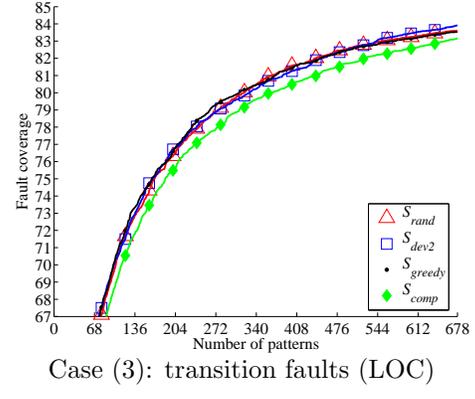
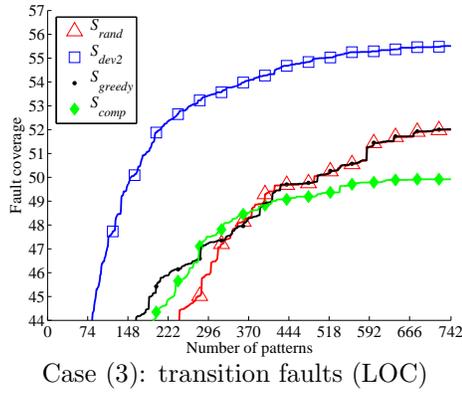
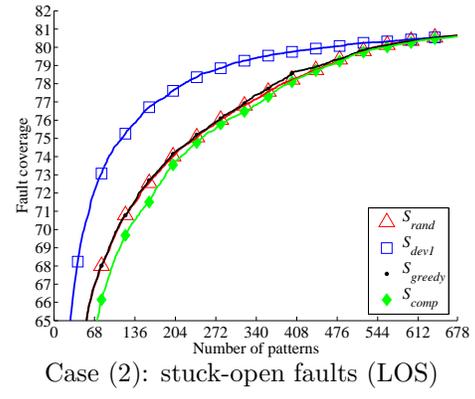
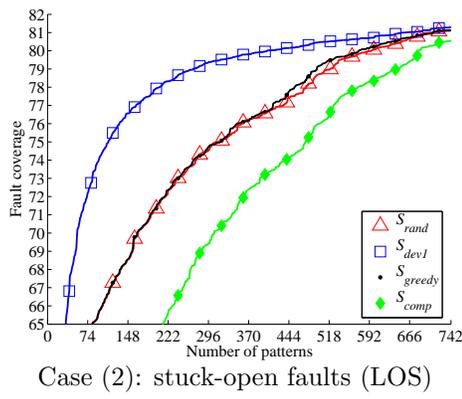
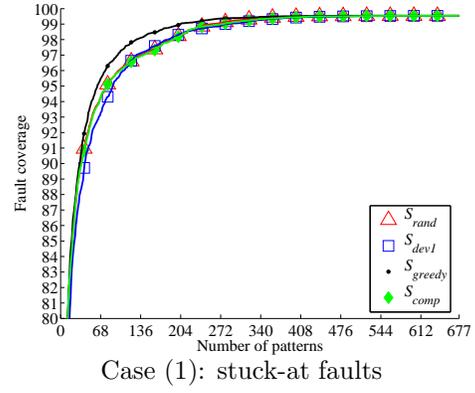
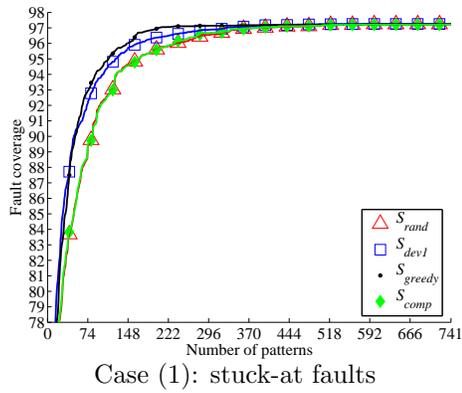
(a) s5378



Case (3): transition faults (LOC)

(b) s9234

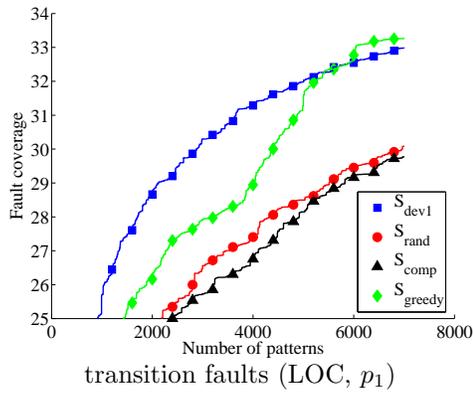
Figure 2.7: Fault coverage for s5378 and s9234 (5-detect stuck-at test cubes): (1) stuck-at faults; (2) stuck-open faults (LOS); (3) transition faults (LOC).



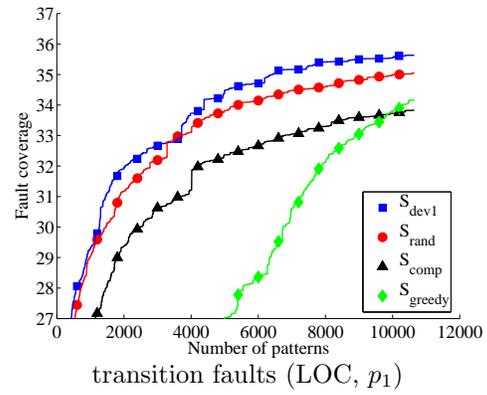
(a) s15850

(b) s38417

Figure 2.8: Fault coverage for s15850 and s38417 (5-detect stuck-at test cubes): (1) stuck-at faults; (2) stuck-open faults (LOS); (3) transition faults (LOC).

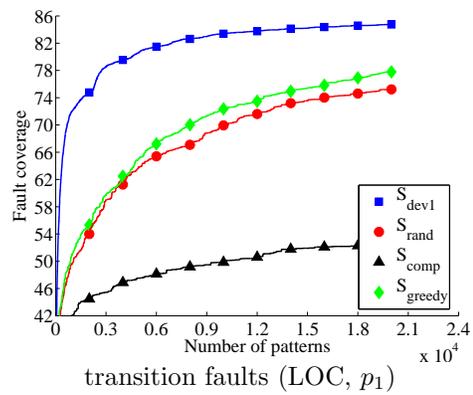


(a) tv80

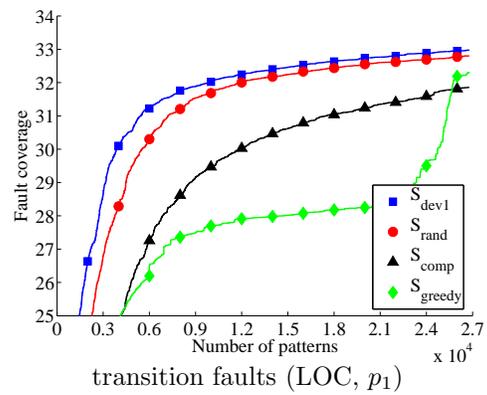


(b) mem_ctrl

Figure 2.9: Fault coverage for tv80 and mem_ctrl (1-detect stuck-at test cubes): transition faults (LOC, p_1).

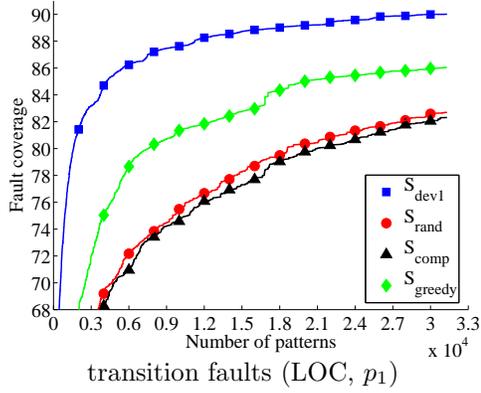


(a) ac97_ctrl

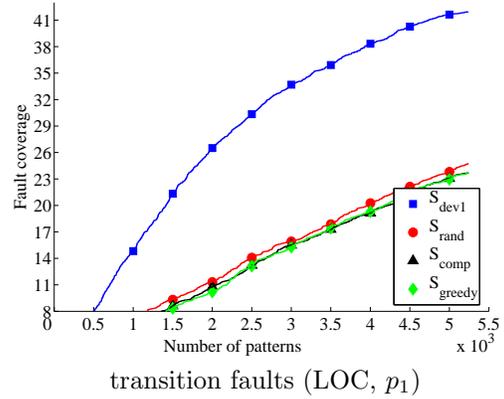


(b) DMA

Figure 2.10: Fault coverage for ac97_ctrl and DMA (1-detect stuck-at test cubes): transition faults (LOC, p_1).

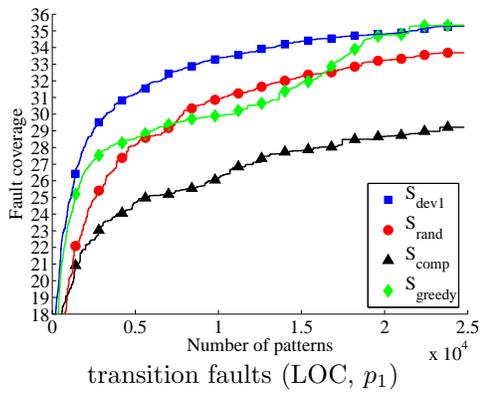


(a) pci_bridge32

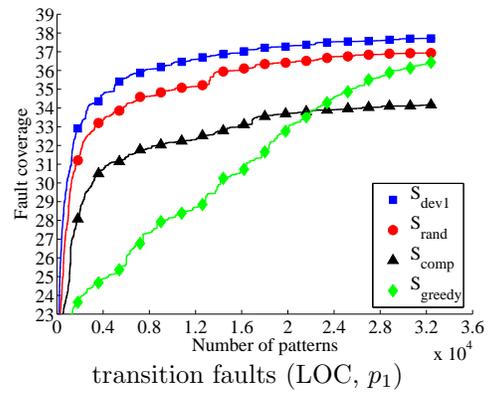


(b) ethernet

Figure 2.11: Fault coverage for pci_bridge32 and ethernet (1-detect stuck-at test cubes): transition faults (LOC, p_1).



(a) tv80



(b) mem_ctrl

Figure 2.12: Fault coverage for tv80 and mem_ctrl (3-detect stuck-at test cubes): transition faults (LOC, p_1).

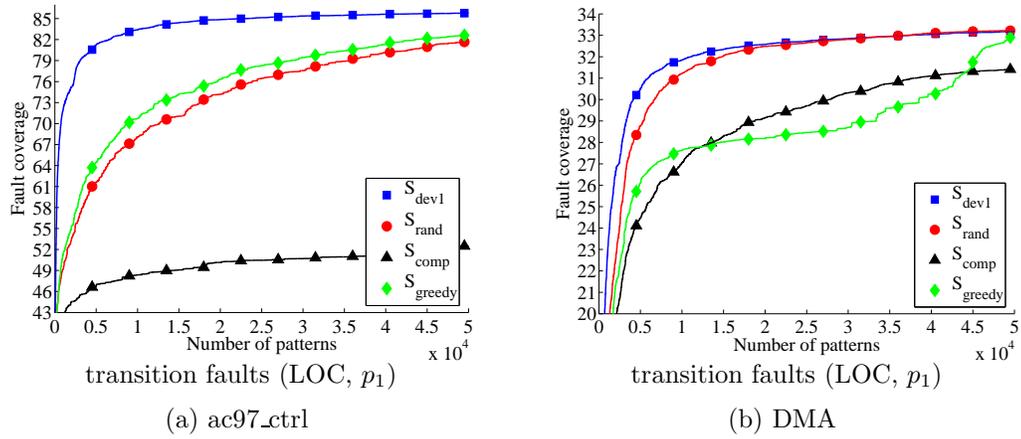


Figure 2.13: Fault coverage for ac97_ctrl and DMA (3-detect stuck-at test cubes): transition faults (LOC, p_1);

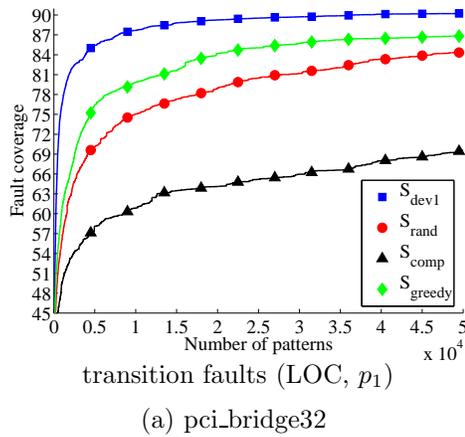


Figure 2.14: Fault coverage for pci_bridge32 (3-detect stuck-at test cubes): transition faults (LOC, p_1).

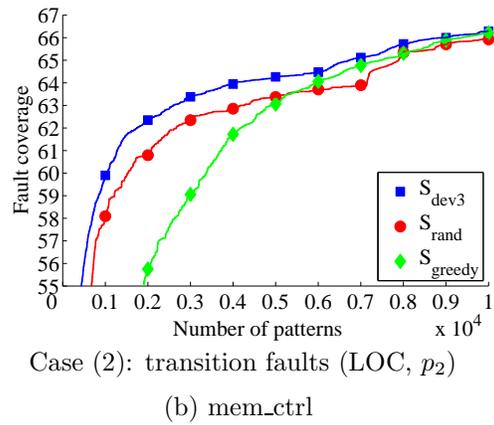
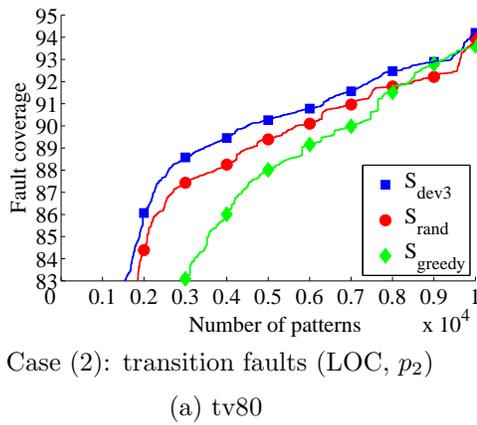
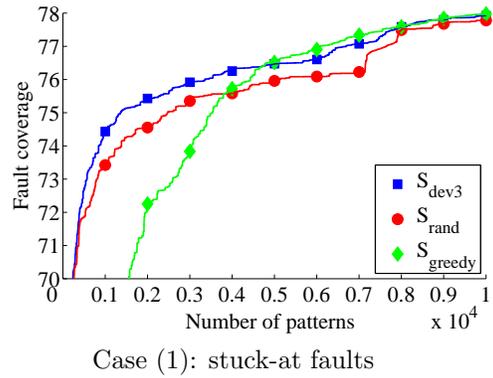
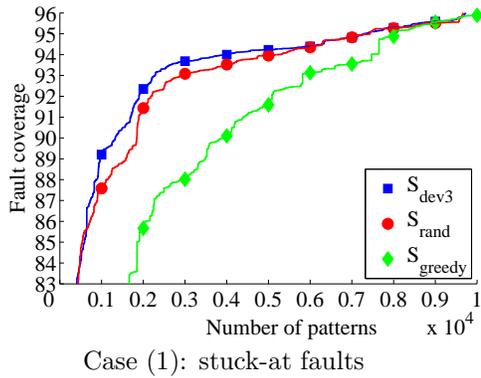


Figure 2.15: Fault coverage for tv80 and mem_ctrl (1-detect transition test cubes): (1) stuck-at faults; (2) transition faults (LOC, p_2).

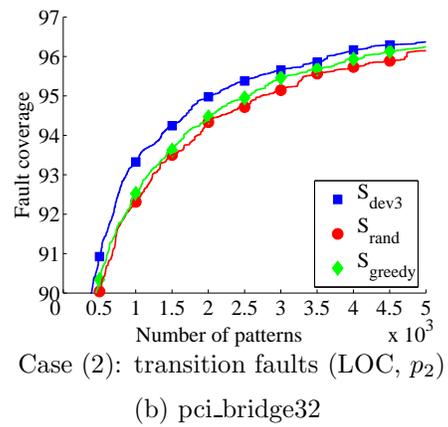
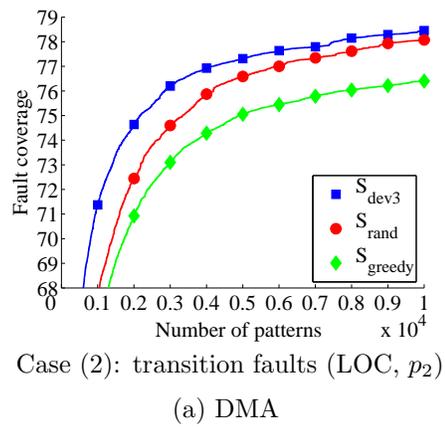
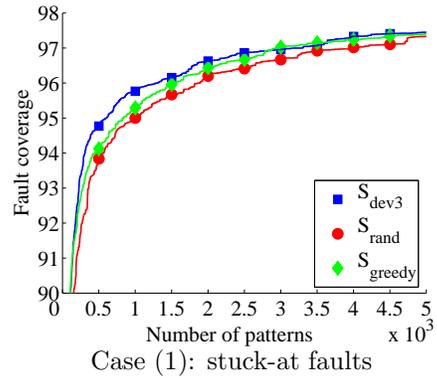
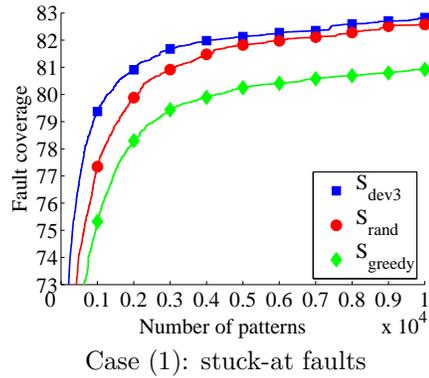


Figure 2.16: Fault coverage for DMA and pci_bridge32 (1-detect transition test cubes): (1) stuck-at faults; (2) transition faults (LOC, p_2).

2.3 Seed Augmentation for LFSR Reseeding

It has been demonstrated in 2.2 that seed-selection based on output deviations provides higher coverage and steeper coverage ramp-up for unmodeled defects compared to several other seed-selection methods. In this section, we present a design-for-testability (DFT) technique for increasing the effectiveness of LFSR reseeding for unmodeled defects [67]. The proposed method relies on seed selection using the output-deviations metric and the on-chip augmentation of seeds using simple bit-operations. Simulation results for benchmark circuits show that compared to LFSR reseeding using output deviations alone, the proposed method provides higher coverage for transition-delay and bridging faults, and steeper coverage ramp-up for these faults for the same number of seeds. For the same pattern count (and much fewer seeds), the proposed method provides comparable unmodeled defect coverage. In all cases, complete coverage of modeled stuck-at faults is obtained. We therefore conclude that high test quality can be obtained with the proposed LFSR reseeding method using a smaller number of seeds.

2.3.1 Bit-Operations-Based Seed Augmentation

We first show how bit-operations on seeds can be combined with deviation-based seed selection. The augmented seed set is evaluated using deviations and we discard the generated seeds that lead to low-deviation patterns.

The bit-operations-based augmentation method consists of four steps. First, given stuck-at test cubes, the original seed set S_{dev} is obtained using the deviation-based seed selection method from [24]. The seeds in S_{dev} are individually labeled as a_1, a_2, \dots during the augmentation process. Next, we apply five bit-operations to each seed in S_{dev} so that five more seeds are generated; these seeds are called the derived seeds. Additional bit operations can also be considered. However, we limit ourselves to

these five simple operations to demonstrate the effectiveness of seed augmentation.

The bit-operations are as follows:

- *Reversing bits.* Here, “reversing” means that we complement for each bit of the seed. For example, the seed 010011 is obtained after performing the “reversing bits” operation on the original seed 101100.
- *Left-rotate by one bit.* For example, the seed 011001 can be derived from 101100.
- *XORing adjacent bits:* given the original seed $O_1O_2\dots O_n$, the derived seed $G_1G_2\dots G_n$ is given by:

$$G_i = \begin{cases} O_i \oplus O_{i+1} & i = 1, 2, \dots, n - 1 \\ O_i \oplus 0 & i = n \end{cases} \quad (2.1)$$

- *XNORing adjacent bits:* given the original seed $O_1O_2\dots O_n$, the derived seed $G_1G_2\dots G_n$ is given by:

$$G_i = \begin{cases} \overline{O_i \oplus O_{i+1}} & i = 1, 2, \dots, n - 1 \\ O_i \oplus 1 & i = n \end{cases} \quad (2.2)$$

- *Interchanging adjacent bits.* For example, the seed 011001 can be derived from 100110.

An additional augmentation method is based on pseudorandom perturbations of a seed, which can be easily implemented using an on-chip LFSR and appropriate biasing logic. In the third step, we obtain the seed set S_{merge} using the following procedures:

- Initialization: Initialize S_{merge} to be the null set;
- Seed selection: Place seed a_1 in S_{merge} and calculate its deviation D_1 . If four of the five patterns derived from a_1 's augmented seeds (through the five bit-operations discussed above) have high deviation values, that is, more than a

fraction θ of D_1 , we place these seeds in S_{merge} . Otherwise, we discard these five generated seeds. Without loss of generality, we set the value of θ to 95% in this Chapter.

We repeat the above steps for a_2, a_3, \dots until the number of seeds in S_{merge} equals the number of seeds in S_{dev} . We expect the patterns derived from S_{merge} to be more effective in detecting unmodeled defects since they have high deviations.

Finally, top-off ATPG is performed to obtain additional test cubes since patterns derived from S_{merge} do not provide complete stuck-at fault coverage. For these test cubes, deviation-based seed selection is used to obtain the seed set S_{topoff} . The final seed set S_{final} is simply $S_{merge} \cup S_{topoff}$.

2.3.2 Hardware Implementation

This section presents the hardware implementation of bit-operation-based LFSR re-seeding.

Architecture for Bit-Operation-Based LFSR Reseeding

Figure 2.17 shows the architecture for the proposed bit-operation-based LFSR re-seeding method. The finite-state-machine (FSM) is used to control the operation of seed loading and seed transformation. For each test pattern, a seed is stored on the tester and loaded into buf1. Under the control of FSM, the Seed Generator unit performs the corresponding seed transformation (the five bit-operations defined in Section 2.3.1) for the seed in buf1. The generated seed is stored in buf2 and loaded into the LFSR. The LFSR then runs in autonomous mode for a number of cycles equal to the length of the longest scan chain.

The “Control” signal is used to indicate whether seed transformation must be carried out. When “Control” is 1, it means that at least four of the generated seeds

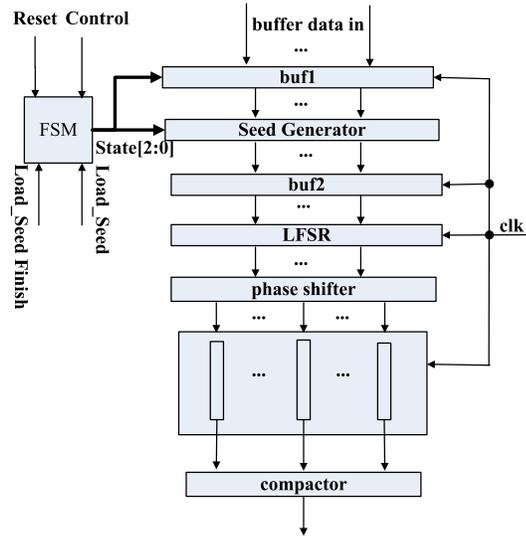


Figure 2.17: Architecture for bit-operation-based LFSR reseeding.

will lead to high-deviation patterns, thus seed transformation will be carried out. Otherwise, seed transformation is not carried out.

Detailed Hardware Implementation

The major components in the hardware implementation are the FSM and the Seed Generator unit.

Interface of the FSM

Table 2.8 lists the major interface of the FSM. Among all these input and output signals, only one extra pin is needed for the “Control” signal. It is used to record whether or not to transform the original seed. $State[2 : 0]$ represents the present state of the FSM. It is used to control the loading and transformation of seeds. “Load_Seed” signal and “Load_Seed_finish” signal are both obtained from the control logic for traditional LFSR reseeding. “Load_Seed” enables the seed buffer buf1 to load a seed from the ATE. “Load_Seed_finish” implies that the loading for the current seed has finished.

State Transition Diagram of FSM Figure 2.18 shows the state transition diagram for the FSM. Under the control of the global “Reset” signal, the FSM enters

Table 2.8: Interface of the FSM.

Signal name	IN/OUT	Function
Reset	IN	Global reset. No need extra pin.
Control	IN	Extra pin. Control whether or not to transform seed.
State[2:0]	OUT	Control the loading of buf1 and the type of bit-operation for Seed Generator.
Load_Seed	IN	Begin loading of seed from ATE.
Load_Seed_finish	IN	Loading for current seed has finished.

the IDLE state. Once “Load_Seed” is set to 1, the FSM jumps to the Load_Buf1 state and beginning to load a seed from the ATE. When “Load_Seed_Finish” is detected to be 1, which means the loading of current seed has been finished, the FSM enters the DEV state. In the DEV state, the LFSR processes the original seed selected by the deviation-based method. When the LFSR has run for enough cycles in the autonomous mode to fill the scan chains with a test pattern derived from the seed in buf2, “Load_Seed” is set to 1, which means that the next seed can be processed by the LFSR. When “Control” is 0, it implies that no transformation should be performed for the current seed and the next seed is from the ATE. The FSM enters the Load_Buf1 state in this case. When “Control” is 1, it means that the transformation will be performed for the current seed. The FSM enters the NOT state in this case. In this state, the LFSR processes the generated seed by reversing the bits of the original seed. When the LFSR finishes filling the scan chains with the current test pattern derived from the seed, the FSM enters the XOR state. Similarly, the FSM enters the XNOR, INTERCHANGE and RO LEFT states in consecutive order. In these states, the LFSR processes the generated seed by the corresponding bit-operation. When “Load_Seed” is set to 1 in RO LEFT state, the FSM enters the Load_Buf1 state and begins to load a new seed from the ATE.

We use 3-bit signals to encode the 8 states of the FSM. Table 2.9 shows the value

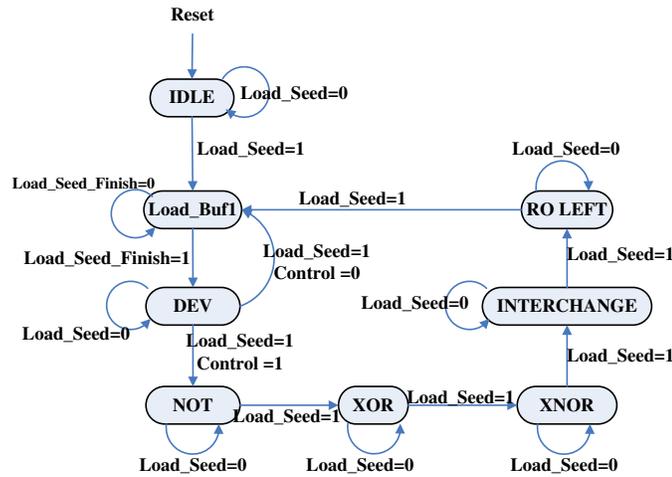


Figure 2.18: State transition diagram of the FSM.

Table 2.9: States of the FSM.

Name	State[2:0]	Name	State[2:0]
IDLE	000	XOR	110
Load_Buf1	011	XNOR	111
DEV	100	INTERCHANGE	001
NOT	101	RO LEFT	010

of State[2:0] for corresponding state. We developed a Verilog model for the FSM and synthesized it using Synopsys Design Compiler.

Logic Design for Seed Generator

The Seed Generator unit implements the five seed transformations for the original seed. We illustrate the logic design of the Seed Generator for 4-bit seeds in Figure 2.19. Signals in1-in4 represent the seed from buf1. Signals out1-out4 represent the seed after bit-operations. The bit-operation logic is implemented by gate types NOT, XOR, XNOR and MUX. State[2:0] provides selection control for MUXes.

Impact on Hardware Overhead and Test Time

We next analyze the hardware overhead and compare the test time to that for traditional LFSR reseeding.

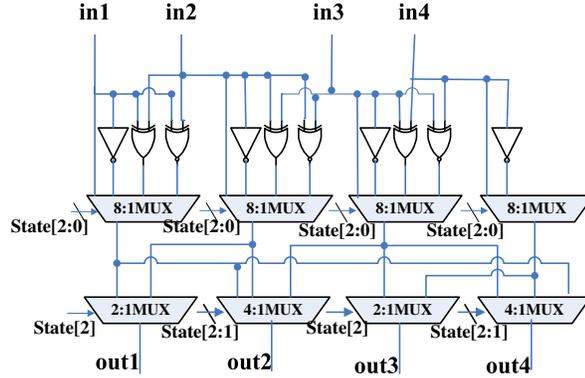


Figure 2.19: Logic design for Seed Generator (example of 4-bit seeds).

Hardware Overhead Suppose the size of LFSR is L bits. Thus, the length of seed is L . For the L -bit seed, we need L NOT gates, $L - 1$ XOR gates, $L - 1$ XNOR gates, L 8:1 multiplexes, $\lfloor L/2 \rfloor$ 4:1 multiplexes, $\lceil L/2 \rceil$ 2:1 multiplexes to implement the Seed Generator unit. We need 2 AND gates, 3 NOT gate, 1 NAND gate, 4 NOR gates, 3 multiplexes, 2 AO4 gates, 1 AO6 gate and 1 AO7 gate to implement the combination part of the FSM. Also, we need $L + 3$ additional D-Flip-Flops (DFFs). A total of L DFFs are needed for buf1 to store the seed loaded from ATE. A total of 3 DFFs and 17 gates are needed for the FSM. One extra pin is needed for “Control” signal, which is used to control whether to transform the current seed.

Comparison of Test Time

In the conventional LFSR reseeding method, when the LFSR is run in the autonomous mode to fill the scan chains with a test pattern, the seed corresponding to the next test pattern is shifted into the on-chip seed buffer. This is done in such a way that the next seed is ready by the time the LFSR finishes filling the scan chains with the current test pattern. Suppose the length of seed is L , the number of pins used to load seeds from ATE is n , and the maximum length of scan chains is C_{max} . The goal is to choose n such that $L/n \leq C_{max}$. If this condition is satisfied, the filling of scan chains will overlap with the loading of seeds from the ATE (except for

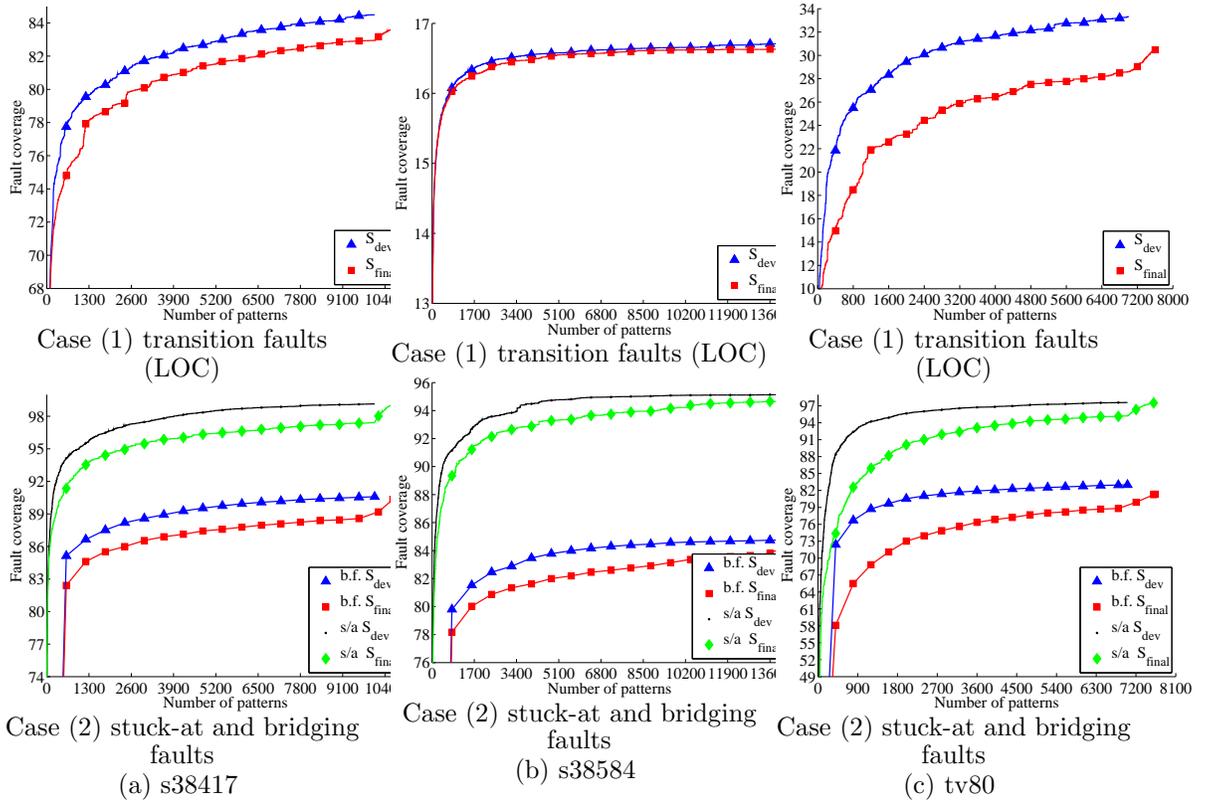


Figure 2.20: Fault coverage ramp-up for s38417, s38584 and tv80 for Experiment 1: (1) transition faults (LOC); (2) stuck-at faults and bridging faults.

the loading of the first seed).

In the proposed method, in order to ensure that the next seed is ready by the time the LFSR finishes filling the scan chains with the current test pattern, we need to choose n , such that $L/n + 1 \leq C_{max}$ (we need one clock cycle to transform a seed based on one type of bit-operation). If this condition is satisfied, to apply the same number of patterns, the test-time overhead will be only one clock cycle (to have the first seed ready in buf2) compared to the conventional method.

2.3.3 Experimental Evaluation

In order to evaluate the effectiveness of our method, we compare the cumulative stuck-at fault coverage, transition fault coverage under launch-on-capture (LOC) (LOC

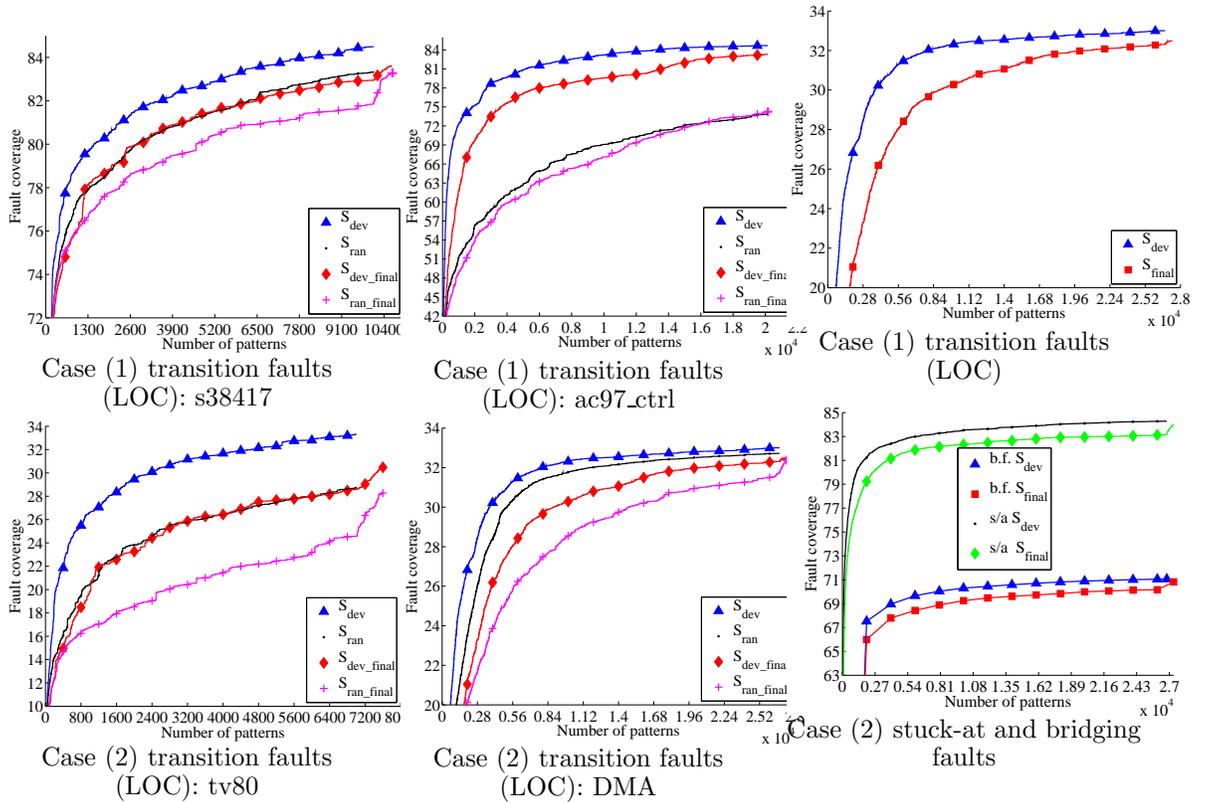


Figure 2.21: LOC transition fault coverage ramp-up for Experiment 3: (1) s38417; (2) tv80.

Figure 2.22: LOC transition fault coverage ramp-up for Experiment 3: (1) ac97_ctrl; (2) DMA.

Figure 2.23: Fault coverage ramp-up for DMA for Experiment 1: (1) transition faults (LOC); (2) stuck-at faults and bridging faults.

is more popular than launch-on-shift (LOS) scheme in industry, hence we do not consider LOS), and static bridging-fault coverage for patterns derived from different seed sets. The experiments are performed on two largest ISCAS-89 circuits and four IWLS-05 benchmarks [68]. Details about the IWLS-05 circuits can be found in [69].

Experimental Setup

All experiments were performed on a 64-bit Linux server with 4 GB memory. The test cubes were generated using a commercial ATPG tool, which was also used to run fault simulation. The program for deviation calculation was coded in C++.

Experiment 1: Same Number of Patterns as with Stand-Alone Deviation Method

We obtained the cumulative stuck-at, transition and static bridging fault coverage of patterns for different seed sets and the same number of patterns. We also compared the storage requirement of the original seed set (selected by the deviation-based method [24]) and the seed set S_{final} obtained using the proposed method. Other seed-selection methods are not considered here because they were shown in [24] to be less effective for unmodeled defects.

Cumulative Stuck-at, Transition Delay and Static Bridging Fault Coverage For each benchmark, three cases are considered: (i) stuck-at faults; (ii) transition faults detected by the LOC scheme; (iii) static bridging faults. The cumulative fault coverage for these cases are obtained by patterns derived from the different seed sets. The static bridging-fault coverage is measured using the modified bridging-coverage estimate (BCE^+) [70]. Note that complete stuck-at fault coverage is obtained using the pattern derived from both S_{dev} and S_{final} .

Figure 2.20 and Figure 2.23 show the fault coverage obtained for different fault models for the same pattern counts. It can be seen from all these cases that the pat-

terns derived from S_{final} provide comparable transition delay and static bridging-fault coverage, since the coverage curves of these patterns are very close to the coverage curve of the patterns derived from S_{dev} . Even though the ramp-up is slightly less steep here, the coverage of the final complete test set is nearly as high as for [24]. Moreover, the complete set of patterns derived from S_{final} provides the same stuck-at fault coverage as that from S_{dev} .

Storage Requirement A key advantage of our method is the reduction in needed for the seeds storage. We can see that the number of bits stored on the tester is significantly reduced for the proposed method, since most of the seeds in S_{merge} are generated on-chip from the original seeds. Table 2.10 shows the storage requirement for S_{dev} and S_{final} . For example, for benchmark *tv80*, 7008 seeds need to be stored on the tester for the deviation-based method from [24], while our approach only requires 1965 seeds to provide the same stuck-at fault coverage. The percentage reduction in seed storage is 72% for *tv80*.

Table 2.10: Comparison of storage requirement (number of seeds).

Benchmark	$ S_{dev} $	$ S_{merge} $	$ S_{topoff} $	$ S_{final} $	<i>Reduction</i> (%)
s38417	10081	1741	564	2305	77%
s38584	14161	2391	133	2524	82%
tv80	7008	1338	627	1965	72%
mem_ctrl	10649	2314	787	3101	71%
ac97_ctrl	20146	3566	8	3674	82%
DMA	26782	4717	565	5282	80%

The above results show that S_{final} can provide namely the same defect coverage as S_{dev} with the same test length (pattern count), but it requires much less test memory for seed storage.

Experiment 2: Same Number of Seeds

Next we examine the fault coverage obtained with the proposed method using the same number of seeds as in [24]. We are able to apply a larger number of patterns because additional seeds are generated on-chip. Figures 2.24 show that significantly higher coverage ramp-up is obtained due to the on-chip augmentation of the loaded seeds.

Experiment 3: Perturbations on Seeds that Lead to High-Deviation and Low-Deviation Patterns

In the proposed method, S_{dev} is used as the original seed set for bit-operations. It is obtained by deviation-based seed selection method and is ensured to derive high-deviation patterns. In order to see whether the results are sensitive to the original seed set, we also perform experiments starting from seed sets that lead to patterns with low deviation. We ensure that the modeled (stuck-at) fault coverage is not affected, i.e., the selected seeds provide the same stuck-at coverage as before. This seed set can be obtained by randomly choosing one viable seed for each test cube, and the set is labeled S_{ran} . For S_{ran} , we repeat the same procedure as for S_{dev} . The sets S_{dev_final} and S_{ran_final} are the final seed sets obtained for the two cases.

Same Number of Patterns Figure 2.21 and Figure 2.22 show the LOC transition fault coverage obtained for the same pattern counts. It can be seen from all these cases that the patterns derived from S_{dev_final} provide higher transition delay coverage than patterns derived from S_{ran_final} . Moreover, patterns from S_{dev_final} yield steeper ramp-up curve than patterns from S_{ran_final} . We have also found that the number of seeds in S_{dev_final} is nearly equal to the number of seeds in S_{ran_final} .

Same Number of Seeds Next we compare the LOC transition fault coverage for the two starting points for the same number of seeds. Figure 2.25 show that higher

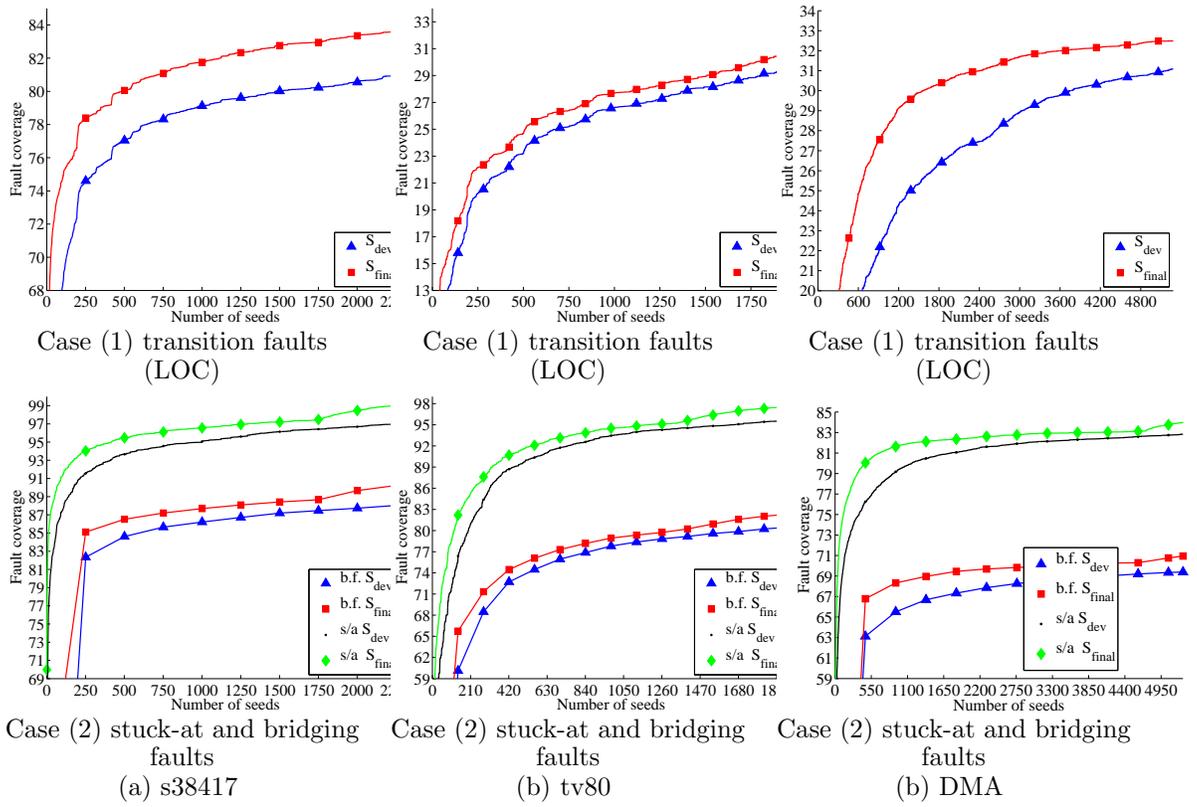


Figure 2.24: Fault coverage ramp-up for s38417, tv80 and DMA for Experiment 2: (1) transition faults (LOC); (2) stuck-at faults and bridging faults.

coverage ramp-up is obtained if starting from S_{dev} , instead of starting from S_{ran} . It means that for the same number of seeds, higher unmodeled fault coverage is obtained if we start with seeds giving high deviation patterns, compared with seeds yielding low deviation patterns.

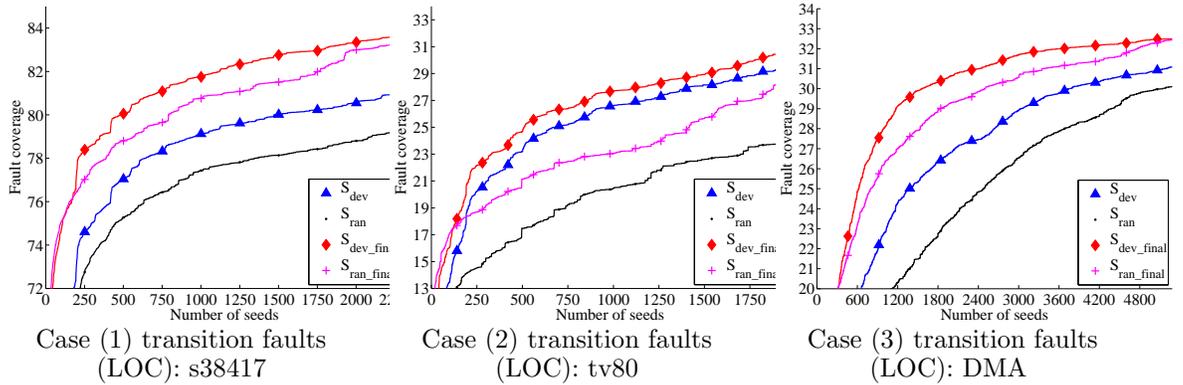


Figure 2.25: LOC transition fault coverage ramp-up for Experiment 3: (1) s38417; (2) tv80; (3) DMA.

2.4 Chapter Summary and Conclusions

In this chapter, we have proposed the use of output deviations as a surrogate coverage metric to select effective seeds for LFSR-reseeding-based test compression. The goal is to select seeds that are expanded to patterns with high output deviations. Experimental results for the ISCAS-89 and IWLS-05 benchmark circuits show that the selected seeds lead to high-deviation patterns, which provide higher defect coverage than patterns derived from other seeds. Moreover, faster fault-coverage ramp-up is obtained using these LFSR seeds. Our experiments are based on both n -detect stuck-at test cubes and 1-detect LOC transition test cubes. For the case of stuck-at test cubes, in order to evaluate defect coverage, we have determined the coverage for transition (stuck-open) faults provided by a set of n -detect stuck-at patterns. For the case of transition test cubes, in order to evaluate defect coverage, we have determined the coverage of stuck-at faults provided by a set of 1-detect transition patterns. We have also shown that the patterns generated using the proposed method have negligible impact on test data volume. We have also highlighted the factors that influence the choice of p_1 and p_2 for the calculation of deviations. This choice is related to the fraction of don't-care bits and the transition-fault coverage provided by the test

cubes.

We have also presented a DFT method for increasing the effectiveness of LFSR reseeding. We have shown how on-chip augmentation of LFSR seeds can be combined with output deviations to ensure that the most effective patterns are applied to the circuit under test. Simulation results have demonstrated that for the same pattern count, the proposed method utilizes significantly fewer seeds, yet provides nearly the same defect coverage ramp-up as a seed-selection method based only on output deviations. For the same number of seeds, the proposed method provides higher coverage and faster coverage ramp-up compared to [24].

Chapter 3

RTL Grading and RTL DFT for Functional Test Sequences

Functional test sequences are often used in manufacturing testing to target defects that are not detected by structural test. Therefore, it is necessary to evaluate the quality of functional test sequences. Moreover, functional test sequences often suffer from low defect coverage since they are mostly derived in practice from existing design-verification test sequences. Therefore, there is a need to increase their effectiveness using design-for-testability (DFT) techniques. In this chapter, we focus on RTL grading and RTL DFT for functional test sequences. We first introduce the preliminaries and calculation of RT-level output deviations. Next we present the grading method for functional test sequences based on the RT-level output deviations. We also present the observation points selection method for an RTL design and a given functional test sequences.

3.1 Output Deviations at RT-level: Preliminaries

In this section, we present basic concepts needed to calculate output deviations at RT-level. Our objective is to use deviations as a surrogate metric for functional test grading.

First, we define the concept of transition count (TC) for a register. Typically, there is dataflow between registers when an instruction is executed and the dataflow affects the values of registers. For any given bit of a register, if the dataflow causes a change from 0 to 1, we record that there is a $0 \rightarrow 1$ transition. Similarly, if the dataflow causes a change from 1 to 0, we record that there is a $1 \rightarrow 0$ transition. If the

dataflow makes no change to this bit of the register, we record that there is a $0 \rightarrow 0$ transition or a $1 \rightarrow 1$ transition, as the case may be.

After a transition occurs, the value of the bit of a register can be correct or faulty (due to an error). When an instruction is executed, there may be several transitions in the bits of a register. Therefore, an error may be manifested after the instruction is executed. We define the output deviation for instruction I_j , $\Delta(j)$, as the probability that an error is produced when I_j is executed. Section 3.2 presents details about the calculation of $\Delta(j)$. The output deviation for an instruction provides a probabilistic measure of the correct operation of instructions at the RT-level.

Similarly, since a functional test sequence is composed of several instructions, we define the output deviation for a functional test sequence to be the probability that an error is observed when this functional test sequence is executed. For example, suppose a functional test sequence, labeled T_1 , is composed of instructions I_1, I_2, \dots, I_N , and let $\Delta(j)$ be the deviation for I_j , as defined above. The output deviation for T_1 , $\Delta^*(T_1)$, is defined as: $\Delta^*(T_1) = \sum_{j=1}^N \Delta(j)$. This corresponds to the probability that an error is produced when T_1 is executed.

Based on these definitions, output deviations can be calculated at RT-level for functional test sequences for a given design. This procedure is described in detail in the next section.

3.2 Deviation Calculation at RT-level

Since our objective is to use deviations as a surrogate metric to grade functional test sequences, we expect test sequences with higher deviation values to provide higher defect coverage. In order to ensure this, we consider three contributors to output deviations. The first is the transition count (TC) of registers. Higher the TC for a functional test sequence, the more likely is it that this functional test sequence will

detect defects. We therefore take TC into consideration while calculating deviations. The second contributor is the observability of a register. The TC of a register will have little impact on defect detection if its observability is so low that transitions cannot be propagated to primary outputs. The third contributor is the amount of logic connected to a register. In order to have a high correlation between RT-level deviation and gate-level stuck-at fault coverage, we need to consider the relationship between RT-level registers and gate-level components.

In this section, the Parwan processor [71] is used as an example to illustrate the calculation of output deviations.

3.2.1 Observability Vector

We first consider the observability of registers. The output of each register is assigned an observability value. The observability vector for a design at RT-level is composed of the observability values of all its registers. Let us consider the calculation of the observability vector for the Parwan processor [71]. The Parwan is an accumulator-based 8-bit processor with a 12-bit address bus. Its architectural block diagram is shown in Figure 3.1(a).

From the instruction-set architecture and RT-level description of Parwan, we extract the dataflow diagram to represent all possible functional paths. Figure 3.1(b) shows the dataflow graph of Parwan. Each node represents a register. The IN and OUT nodes represent memory. A directed edge between registers represents a possible functional path between registers. For example, there is an edge between the AC node and the OUT node. This edge indicates that there exists a possible functional path from register AC to memory.

From the dataflow diagram, we can calculate the observability vector. First, we define the observability value for the OUT node. The primary output OUT has the highest observability since it is directly observable. Using sequential-depth-like

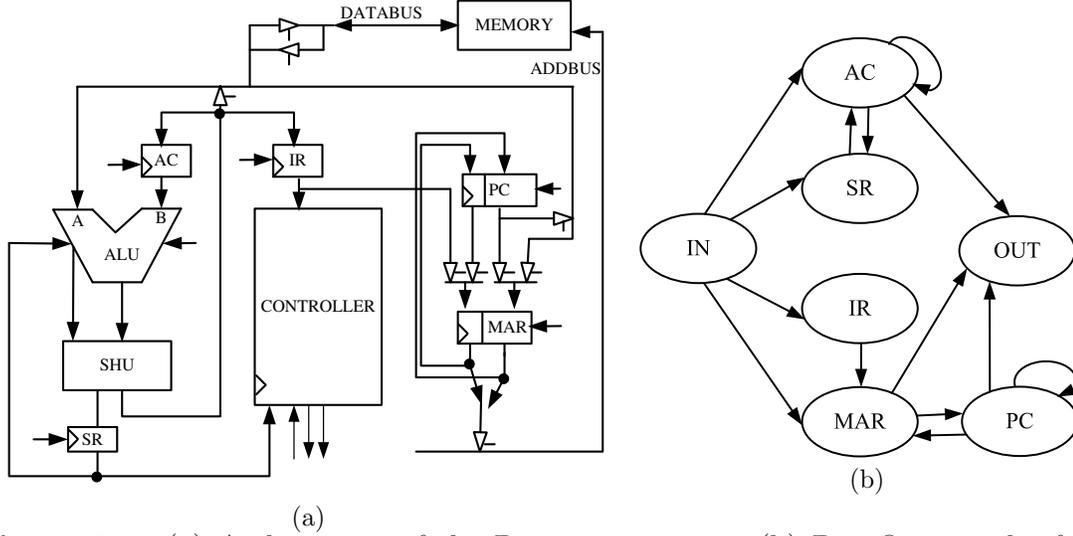


Figure 3.1: (a) Architecture of the Parwan processor; (b) Dataflow graph of the Parwan processor (only the registers are shown).

measure for observability [2], we define the observability value of OUT to be 0, written as $OUT_{obs} = 0$. For every other register node, we define its observability parameter as 1 plus the minimum of the observability parameters of all its fanout nodes. For example, the fanout nodes of register AC are OUT, SR, and AC itself. Thus the observability parameter of register AC is 1 plus the minimal observability parameter among OUT, SR, AC. That is, the observability parameter of register AC is 1. In the same way, we can obtain the observability parameters for MAR, PC, IR and SR. We define the observability value of a register as the reciprocal of its observability parameter. Finally, we obtain the observability vector for Parwan. It is simply $(\frac{1}{AC_{obs}} \frac{1}{IR_{obs}} \frac{1}{PC_{obs}} \frac{1}{MAR_{obs}} \frac{1}{SR_{obs}})$, i.e., $(1 \ 0.5 \ 1 \ 1 \ 0.5)$.

3.2.2 Weight Vector

The weight vector is used to model how much combinational logic a register is connected to. Each register is assigned a weight value, representing the relative sizes of its input cone and fanout cone. The weight vector for a design is composed of the

Table 3.1: Weight vector for registers (Parwan).

	<i>AC</i>	<i>IR</i>	<i>PC</i>	<i>MAR</i>	<i>SR</i>
No. of faults affecting register	2172	338	936	202	2020
Weight value	1	0.1556	0.4309	0.093	0.930

weight values of all its registers. Obviously, if a register has a large input cone and a large fanout cone, it will affect and be affected by many lines and gates. Thus it is expected to contribute more to defect detection. In order to accurately extract this information, we need gate-level information to calculate the weight vector. We only need to report the number of stuck-at faults for each component based on the gate-level netlist. This can be easily implemented without gate-level logic simulation by an automatic test pattern generation (ATPG) tool or a design analysis tool. Here we use Flextest to obtain the number of stuck-at faults for each component in Parwan: there are 248 stuck-at faults in AC, 136 in IR, 936 in PC, 202 in MAR, 96 in SR, 1460 in ALU, and 464 in SHU.

Based on the RT-level description of the design, we can determine the fanin cone and fanout cone of a register. For example, the AC register is connected to three components: AC, SHU and ALU. Given a set of registers $\{R_i\}, i = 1, 2, \dots, n$, let f_i be the total number of stuck-at faults in components connected to register R_i . Let $f_{max} = \max\{f_1, \dots, f_n\}$. We define the weight of register R_i as f_i/f_{max} to normalize the size of gate-level logic. Table 3.1 shows the numbers of faults affecting registers and weights of registers. We can see that $f_{max} = 2172$ for Parwan processor, which is the number of faults in AC. The weight of IR can be calculated as $338/2172$, i.e., 0.1556. In this way, weights of other registers can also be obtained. Finally, we get the weight vector (1 0.1556 0.4309 0.093 0.930).

3.2.3 Threshold value

The higher the TC is for a functional test sequence, the more likely is it that this functional test sequence will detect defects. However, suppose that the TC is already large enough (greater than a threshold value) after the given functional test sequence is executed for a given number of clock cycles. As the functional test sequence is applied for more clock cycles, TC keeps increasing. However, higher values of TC make no significant contribution to the detection of new defects if they generate only a few new transitions in the registers. The threshold value is used to model what value of TC for a register is useful for defect detection. Each register is assigned a threshold value, representing an upper limit on the TC that is useful for defect detection. While different threshold values can be used for various types of transitions of a register, we assume here without loss of generality that all four types of transitions in a register are assigned identical threshold values. We also assume that all registers are assigned identical threshold values.

The threshold value can be obtained by learning from a few of the given functional test sequences. We only need to run logic simulation at RT-level for these test sequences. Suppose that we have k registers in the design. Our goal is to learn from m functional test sequences T_1, T_2, \dots, T_m . The parameter m is a user-defined parameter, and it can be chosen based on how much pre-processing can be carried out in a limited amount of time. In this paper, we use the following steps to obtain the threshold value.

- Run RT-level Verilog simulation for the selected m functional test sequences. Record the new transition counts on all registers for each clock cycle during the execution of the test sequence;
- For each of the m functional test sequences, plot the cumulative new transition

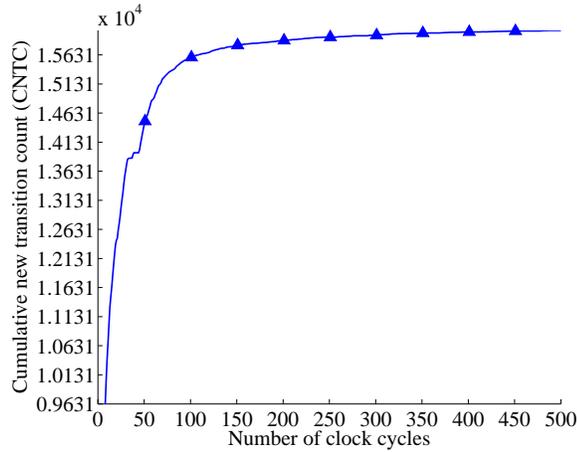


Figure 3.2: An example to illustrate cumulative new transition counts.

count (CNTC). Figure 3.2 is shown as an example. The x-axis represents the number of clock cycles. The y-axis shows the CNTC, summed over all registers. Our experiments show that this generic plot is representative of benchmark circuits and functional test sequences.

- From the above figure, find the “critical point” on the x-axis, after which the CNTC curve flattens out. Formally speaking, the *critical point* is the x-axis value at which the CNTC reaches 98% of the maximum CNTC value. For example, from Figure 3.2, we can see that the critical point is the 130th clock cycle. Denote the critical point for m test sequences as the set $\{cp_1, \dots, cp_m\}$;
- Obtain the aggregated TC for all registers up to the critical point. Denote the aggregated TC for m test sequences by the vector $(th_1 \ th_2 \ \dots \ th_m)$. Given k registers in the design, we set the threshold value th to be the average $(th_1/k + \dots th_m/k)/m$.

3.2.4 Calculation of Output Deviations

The output deviations can be calculated for functional test sequences using the TCs, the observability vector, the weight vector and the threshold value. First, we define the countable TCs for registers. Suppose the $0 \rightarrow 0$ TC of register R_i for instruction I_j is recorded as t_{ij00} . The countable $0 \rightarrow 0$ TC of register R_i for instruction I_j is defined as follows:

$$t_{ij00}^* = \begin{cases} 0 & (\sum_{p=1}^{j-1} t_{ip00} \geq th) \\ th - \sum_{p=1}^{j-1} t_{ip00} & (\sum_{p=1}^{j-1} t_{ip00} < th < \sum_{p=1}^j t_{ip00}) \\ t_{ij00} & (\sum_{p=1}^j t_{ip00} \leq th) \end{cases}$$

Here th is the threshold value for the design.

Suppose TS is composed of q instructions I_1, I_2, \dots, I_q . For each instruction I_j , suppose the countable $0 \rightarrow 0$ TC of register R_i ($1 \leq i \leq k$) for instruction I_j is t_{ij00}^* , the countable $0 \rightarrow 1$ TC of register R_i for I_j is t_{ij01}^* , the countable $1 \rightarrow 0$ TC of register R_i for I_j is t_{ij10}^* , and the countable $1 \rightarrow 1$ TC of register R_i for I_j is t_{ij11}^* . Given the threshold value th , the output deviation $\Delta(j)$ for instruction I_j can be calculated by considering all possible countable transitions and the different registers:

$$\begin{aligned} \Delta(j) = \frac{1}{k} \cdot \frac{1}{4 \cdot \max_{i=1}^k \{th \cdot o_i \cdot w_i\}} & \left(\sum_{i=1}^k t_{ij00}^* \cdot o_i \cdot w_i + \right. \\ & \left. \sum_{i=1}^k t_{ij01}^* \cdot o_i \cdot w_i + \sum_{i=1}^k t_{ij10}^* \cdot o_i \cdot w_i + \sum_{i=1}^k t_{ij11}^* \cdot o_i \cdot w_i \right) \end{aligned} \quad (3.1)$$

In Equation (4.1), $\sum_{i=1}^k t_{ij00}^* \cdot o_i \cdot w_i$ represents the contribution of countable $0 \rightarrow 0$ transitions of all registers. Similarly, $\sum_{i=1}^k t_{ij01}^* \cdot o_i \cdot w_i$, $\sum_{i=1}^k t_{ij10}^* \cdot o_i \cdot w_i$, and $\sum_{i=1}^k t_{ij11}^* \cdot o_i \cdot w_i$ represent the contribution of countable $0 \rightarrow 1$, $1 \rightarrow 0$ and $1 \rightarrow 1$ transitions of all registers, respectively. These contributions are normalized by $4 \cdot \max_{i=1}^k \{th \cdot o_i \cdot w_i\}$.

Based on the deviation definition in Section 3.1, the deviation for TS can be calculated as

$$\begin{aligned}
\Delta^*(TS) &= \sum_{j=1}^q \Delta(j) \\
&= \frac{1}{4k \cdot \max_{i=1}^k \{th \cdot o_i \cdot w_i\}} \cdot \\
&\quad \sum_{j=1}^q \sum_{i=1}^k \{(t_{ij00}^* + t_{ij01}^* + t_{ij10}^* + t_{ij11}^*) \cdot o_i \cdot w_i\} \\
&= \frac{1}{4k \cdot \max_{i=1}^k \{th \cdot o_i \cdot w_i\}} \cdot \\
&\quad \sum_{i=1}^k \left\{ \sum_{j=1}^q \{(t_{ij00}^* + t_{ij01}^* + t_{ij10}^* + t_{ij11}^*) \cdot o_i \cdot w_i\} \right\}
\end{aligned} \tag{3.2}$$

Let $S_{i00} = \sum_{j=1}^q \{t_{ij00}^* \cdot o_i \cdot w_i\}$, $S_{i01} = \sum_{j=1}^q \{t_{ij01}^* \cdot o_i \cdot w_i\}$, $S_{i10} = \sum_{j=1}^q \{t_{ij10}^* \cdot o_i \cdot w_i\}$, and $S_{i11} = \sum_{j=1}^q \{t_{ij11}^* \cdot o_i \cdot w_i\}$, Equation (3.2) can now be written as

$$\begin{aligned}
\Delta^*(TS) &= \frac{1}{4k \cdot \max_{i=1}^k \{th \cdot o_i \cdot w_i\}} \cdot \\
&\quad \sum_{i=1}^k \{S_{i00} + S_{i01} + S_{i10} + S_{i11}\}
\end{aligned} \tag{3.3}$$

Note that S_{i00} represents the aggregated contributions of countable $0 \rightarrow 0$ TC of register R_i for all the instructions in TS . The parameters S_{i01} , S_{i10} , and S_{i11} are defined in a similar way.

Let $S_{00}^* = \sum_{i=1}^k S_{i00}$, $S_{01}^* = \sum_{i=1}^k S_{i01}$, $S_{10}^* = \sum_{i=1}^k S_{i10}$, and $S_{11}^* = \sum_{i=1}^k S_{i11}$, Equation (4.2) can be rewritten as:

$$\Delta^*(TS) = \frac{1}{4k \cdot \max_{i=1}^k \{th \cdot o_i \cdot w_i\}} \cdot \{S_{00}^* + S_{01}^* + S_{10}^* + S_{11}^*\} \quad (3.4)$$

Note that S_{00}^* represents the aggregated contributions of countable 0→0 TC of all registers for all the instructions in TS . The parameters S_{01}^* , S_{10}^* , and S_{11}^* are defined in a similar way.

3.3 Functional Test Grading at RT-Level

It is very time-consuming to evaluate the quality of functional test sequences by the traditional gate-level fault simulation. Therefore, in this section, we propose a deviation-based method to grade functional test sequences at RT-level without explicit fault simulation. First, we calculate the RT-level deviations without explicit fault simulation. Next, fault coverage targeting various fault models is obtained by gate-level fault simulation. The effectiveness of the proposed method is evaluated by the correlation between the RT-level deviations and gate-level fault coverage. It is also evaluated by the ramp up curves.

3.3.1 Experimental Evaluation

We evaluate the efficiency of deviation-based test-sequence grading by performing experiments on the Biquad filter core [42] and the Scheduler module of the Illinois Verilog Model (IVM) [72] [73]. The Biquad filter core is an implementation of an infinite impulse response (IIR) filter with two poles and two zeros. It uses the wishbone interface for reading and writing of filter coefficient registers. The coefficient width is set to 64 bits. A synthesized gate-level design for the Biquad filter consists of 160,147 gates and 1116 flip-flops. IVM employs a microarchitecture that is similar in complexity to the Alpha 21264. It has most of the features of modern microprocessors,

featuring superscalar operation, dynamical scheduling, and out-of-order execution. Relevant research based on IVM has recently been reported in [74] [75]. The Verilog model for Scheduler consists of 1090 lines of code. A synthesized gate-level design for it consists of 375,061 gates and 8,590 flip-flops.

Our first goal is to show high correlation between RT-level deviations and gate-level coverage for various fault models. The second goal is to investigate the ramp-up of the gate-level fault coverage for various functional test sequences. These sequences are obtained using different sequence-reordering methods. It is well-known that we cannot accurately evaluate unmodeled defect coverage through simulation using a specific fault model, because this approach relies on the fault model to represent defect behavior. Unmodeled defect coverage can be determined accurately by analyzing silicon data. In this work, we consider various fault models to represent defect behavior and evaluate the correlation between RTL deviations and gate-level coverage for each fault model.

Results for Biquad filter

Experimental setup All experiments for Biquad filter were performed on a 64-bit Linux server with 4 GB memory. Design Compiler (DC) from Synopsys was used to extract the gate-level information for calculating weight vector. Synopsys Verilog Compiler (VCS) was used to run Verilog simulation and compute the deviations. A commercial tool was used to run gate-level fault simulation and Matlab was used to obtain the Kendall's correlation coefficient [43]. The Kendall's correlation coefficient is used to measure the degree of correspondence between two rankings and assessing the significance of this correspondence. It is used in this paper to measure the degree of correlation of RT-level deviations and gate-level fault coverage. A coefficient of 1 indicates perfect correlation while a coefficient of 0 indicates no correlation.

We adopt 6 test sequences, labeled as TS_1, TS_2, \dots, TS_6 , respectively. Each sequence consists of 500 clock cycles of vectors. TS_1, TS_2 and TS_3 are generated according to the functionality of the Biquad filter. The data input in TS_1, TS_2 and TS_3 is composed of samples from a sinusoid signal with 1 KHz frequency, a cosine signal with 15 KHz frequency and white noise. In TS_4, TS_5 and TS_6 , the data input is randomly generated.

Threshold value In order to calculate the RT-level output deviations, we need to set the threshold value. We obtain the threshold value for Biquad filter according to the description in Section 3.2. In this work, we randomly select two test sequences, T_1 and T_4 , for learning the threshold value. For each of them, we run Verilog simulation and draw the figure of cumulative new transition counts, as shown in Figure 3.3. We can obtain the critical points for T_1 and T_4 are 242th, 228th clock cycle respectively. We then record the aggregated TCs for all registers till the critical point for each of the two target test sequences, i.e., (270072 254448). Since there are altogether 17 registers in the Biquad filter, we set the threshold value TH_1 to be $(270072/17+254448/17)/2$, i.e, 15, 427. In the above setting of threshold value, the CNTC of the critical point is 98% of the maximum CNTC. We can also consider other percentages in setting the critical point. For example, for a given percentage of 95%, we obtain the threshold value as 7, 878, labeled as TH_2 . For a percentage of 90%, we obtain the threshold value as 4, 759, labeled as TH_3 .

Correlation between output deviations and gate-level fault coverage For these six functional test sequences, we derived the Kendall’s correlation coefficient between their RT-level output deviations and their gate-level fault coverage. The stuck-at fault model is used for the Biquad filter, as well as the transition fault model, the bridging-coverage estimate (BCE) [77], and the modified BCE measure (BCE^+) [78]. Figure 3.4 presents details about the experimental flow. First we

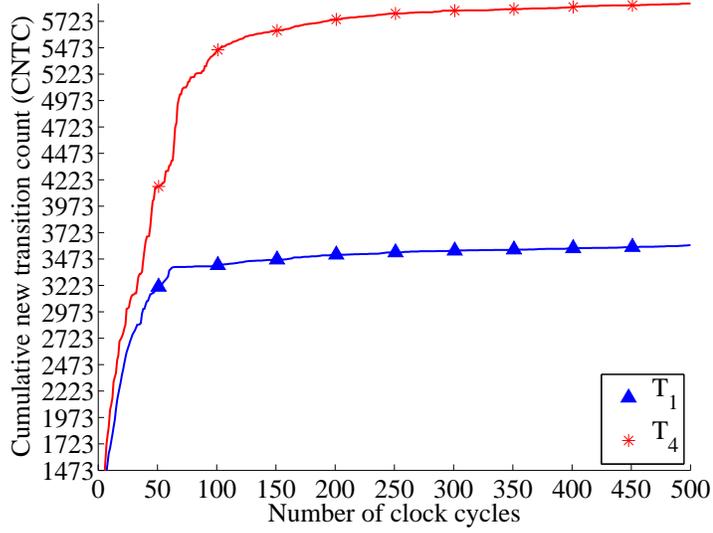


Figure 3.3: Cumulative new transition counts for T_1 and T_4 (Biquad filter).

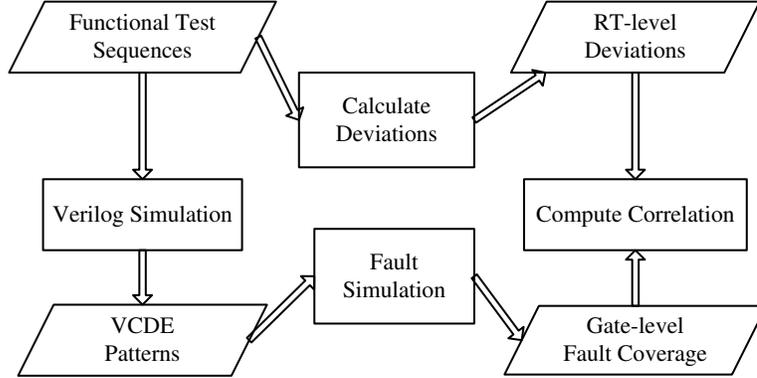


Figure 3.4: Experimental flow.

calculate the deviations for the six functional test sequences according to the method described in Section 3.2 . For each threshold value, we obtain the deviation values for the six functional test sequences and record them as a vector. For example, for the threshold value TH_1 , the deviations are recorded as $DEV_1(dev_1, dev_2, \dots, dev_6)$.

We next obtain test patterns in extended-VCD (VCDE) format for the six functional test sequences by running Verilog simulation. Using the VCDE patterns, gate-level fault coverage is obtained by running fault simulation for the ten functional test sequences. The coverage values are recorded as a vector: $COV(cov_1, cov_2, \dots, cov_6)$.

In order to evaluate the effectiveness of the deviation-based functional test-grading method, we calculate the Kendall's correlation coefficient between DEV_1 and COV . Figure 3.5 shows the correlation between deviations and stuck-at/transition fault coverage, as well as between deviations and BCE/BCE^+ metrics, for different threshold values. For stuck-at faults and bridging faults, we see that the coefficients are very close to 1 for all three threshold values. For transition faults, the correlation is less, but still significant. The results demonstrate that the deviation-based method is effective for grading functional test sequences. The total CPU time for deviation calculation and test-sequence grading is less than 2 hours. The CPU time for gate-level stuck-at (transition) fault simulation is 12 (19) hours, and the CPU time for computing the gate-level BCE (BCE+) measure is 15 (18) hours, thus we are able to reduce CPU time significantly.

Note that we have proposed the use of weight and observability vectors to improve the accuracy of fault grading. If observability and weight vectors are ignored and only transition counts are considered in the calculation of output deviations, the correlation coefficients between output deviations and gate-level fault coverage are

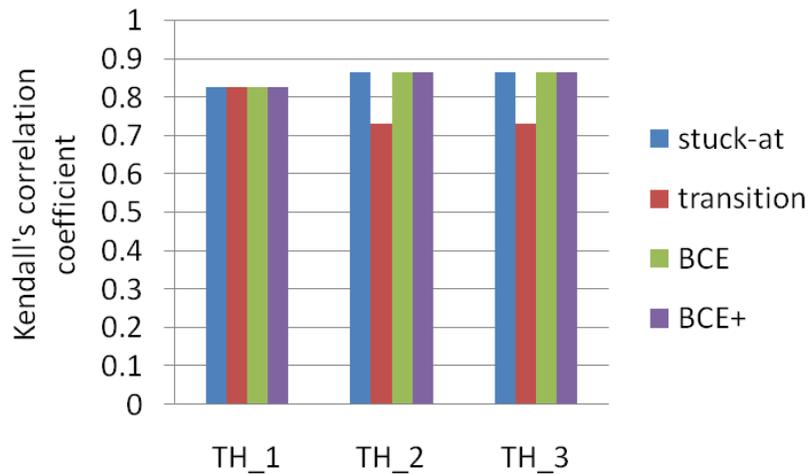


Figure 3.5: Correlation between output deviations and gate-level fault coverage (Biquad).

reduced considerably. These results are shown in Figure 3.6.

Cumulative gate-level fault coverage (Ramp-up) We next evaluate the effectiveness of output deviations by comparing the cumulative gate-level fault coverage of several reordered functional test sequences. For the Biquad filter, traditional stuck-at fault coverage as well as *BCE+* metric are considered.

These reordered sets are obtained in four ways: (i) baseline order TS_1, TS_2, \dots, TS_6 ; (ii) random ordering $TS_1, TS_3, TS_2, TS_6, TS_4, TS_5$; (iii) random ordering $TS_3, TS_5, TS_1, TS_2, TS_4, TS_6$; (iv) the descending order of output deviations. In the deviation-based method, test sequences with higher deviations are ranked first.

Figure 3.7-3.8 show cumulative stuck-at fault coverage, and *BCE+* metric for the four reordered functional test sequences following the above four orders. We can see that the deviation-based method results in the steepest curves for cumulative stuck-at coverage and *BCE+* metric.

Correlation results for longer functional test sequences The above results are obtained based on 6 test sequences, each of which consists of 500 clock cycles.

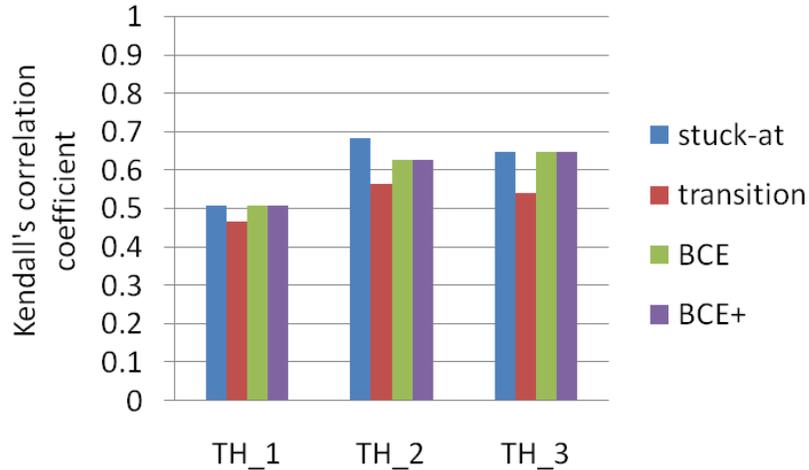


Figure 3.6: Correlation between output deviations (only consider transition counts) and gate-level fault coverage (Biquad).

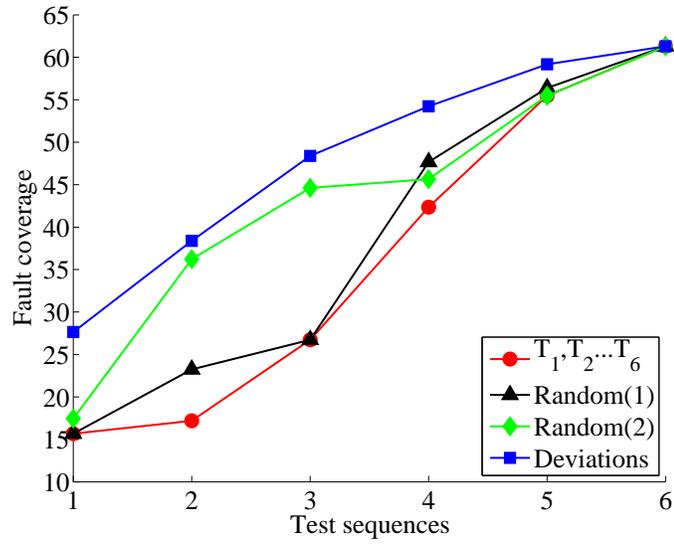


Figure 3.7: Cumulative stuck-at fault coverage for various test-sequence ordering methods (Biquad).

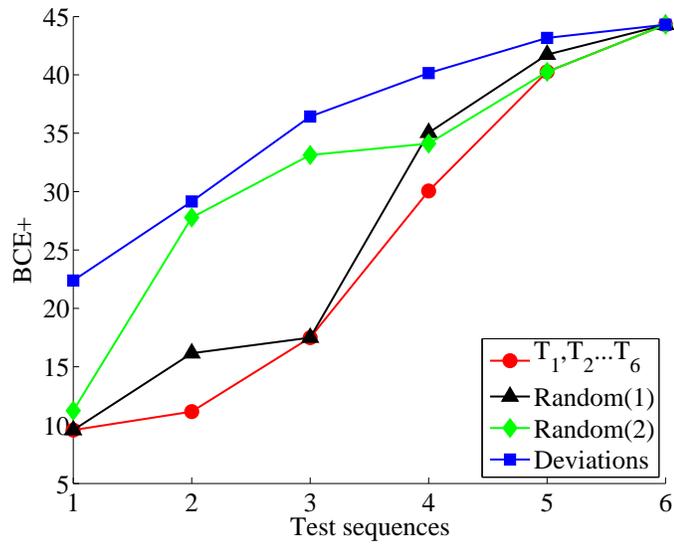


Figure 3.8: Cumulative BCE+ for various test-sequence ordering methods (Biquad).

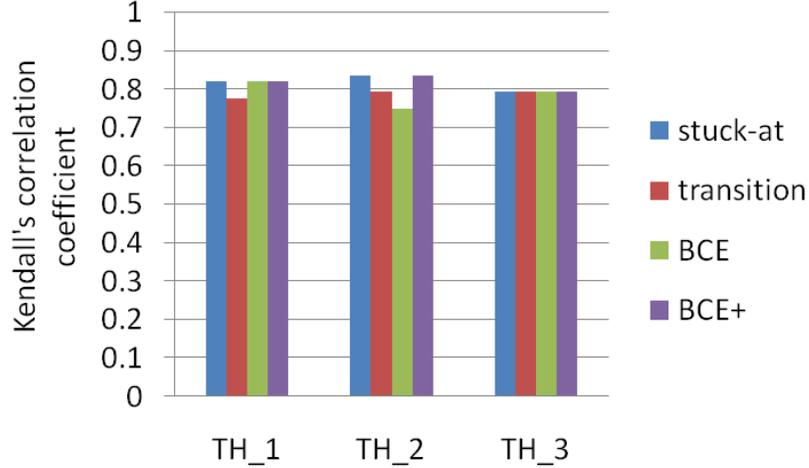


Figure 3.9: Correlation result for longer functional test sequences (Biquad).

To further evaluate the effectiveness of our method, we consider longer functional test sequences. Each test sequence is now 5,000 clock cycles long. The data inputs in TS_1 , TS_2 and TS_3 are obtained by changing the sampling frequency so that they are composed of 5,000 cycles. For TS_4 , TS_5 and TS_6 , the data input is randomly generated.

Figure 3.9 shows the correlation between deviations and stuck-at/transition fault coverage, as well as between deviations and $BCE/BCE+$ metrics, for different threshold values, namely 98%, 95%, and 90%. We can see that the coefficients are still significant for all three threshold values. The results demonstrate that the deviation-based method is effective for grading longer functional test sequences.

Results for Scheduler module

To further demonstrate the benefits of the RT-level deviation metric for larger circuits and longer test sequences, we perform experiments on the Scheduler module.

The experimental setup and procedure are similar to that of Biquad filter: 1) obtain the RT-level deviations by considering TCs, observability vector, weight vector

Table 3.2: Registers and register arrays in the Scheduler module.

Name	Width	Array	Name	Width	Array
sb_counters	3	0 : 79	instout0	222	–
issued	1	0 : 31	instout1	222	–
valid	1	0 : 31	instout2	222	–
issue_head	5	–	instout3	222	–
issue_tail	5	–	instout4	222	–
inst_array	217	0 : 31	instout5	222	–

and threshold value; 2) obtain the gate-level fault coverage; 3) calculate the correlation coefficient and compare the coverage ramp-up curves.

Scheduler module

The Scheduler dynamically schedules instructions, and it is a key component of the IVM architecture [74]. It contains an array of up to 32 instructions waiting to be issued and can issue 6 instructions in each clock cycle.

Table 3.2 shows the registers and register arrays in the Scheduler module. The first and the fourth columns list the name of the register or register array. The second and fifth columns represent the width of each register in bits. The third and sixth columns indicate whether it is a register array. If it is a register array, the index value is shown.

Experimental setup The experimental setup is the same as that for Biquad filter. For the Scheduler module, all experiments are based on 10 functional test sequences. Six of them are composed of instruction sequences, referred to as T_0 , T_1 , T_2 , T_3 , T_4 , T_5 . The other four, labeled as T_6 , T_7 , T_8 and T_9 , are obtained indirectly by sequential ATPG. First, cycle-based gate-level stuck-at ATPG is carried out. From the generated patterns, stimuli are extracted and composed into a functional test sequence, labeled as T_6 . Next we perform “not”, “xnor”, “xor” bit-operations on T_6 separately and obtain new functional test sequences T_7 , T_8 and T_9 . T_7 is obtained

Table 3.3: Weight vector for Scheduler module.

Registers	Fanin count	Fanout count	No. of connections	Weight
sb_counters	9211	4800	14011	0.0183
issued	112932	355464	468396	0.6113
valid	4886	355482	360368	0.4703
issue_head	2986	397086	400072	0.5221
issue_tail	146	202732	202878	0.2648
inst_array	275639	490584	766223	1
instout0	92962	4367	97329	0.1270
instout1	118629	4367	122996	0.1605
instout2	92553	4775	97328	0.1270
instout3	92638	4687	97325	0.1270
instout4	93045	4137	97182	0.1268
instout5	118337	4137	122474	0.1598

by inverting each bit of T_6 ; T_8 is obtained by performing “xnor” operation between the adjacent bits of T_6 ; T_9 is obtained by performing “xor” operation between the adjacent bits of T_6 . The above bit-operations do not affect the clock and reset signals in the design.

RT-level deviations

Using the deviation-calculation method in Section 3.2, we calculate the RT-level deviations for the Scheduler module by considering the parameter TC, weight vector, observability vector and threshold value.

To obtain the weight vector, we first determine the fanin and fanout counts for each register or register array. A net is considered to be in the fanin cone of a register if there is a path through combinational logic from the net to that register. Similarly, a net is considered to be in the fanout cone of a register if there is a path through combinational logic from that register to the net. Table 3.3 lists these counts for each register (array) and shows the weight vector components. In Column 4, “No. of connections” is the sum of the fanin count and the fanout count.

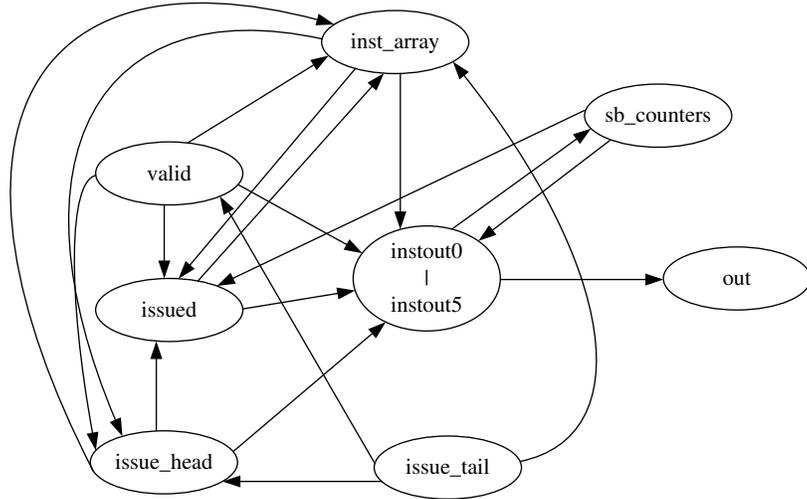


Figure 3.10: Dataflow diagram of Scheduler module.

The observability vector is obtained using the dataflow diagram extracted from the design. The dataflow diagram is extracted automatically. Many logic analysis tools can implement this function. Here, we use Design Compiler of Synopsys and use its features of tracing fanin and fanout registers to construct the dataflow graph. Figure 3.10 shows the dataflow diagram of the Scheduler module.

We consider the following order of the registers: `sb_counters`, `issued`, `valid`, `issue_head`, `issue_tail`, `instout0`, `instout1`, `instout2`, `instout3`, `instout4`, `instout5`, and `inst_array`. The corresponding observability vector is $(0.5 \ 0.5 \ 0.5 \ 0.3333 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0.5)$.

We obtain the threshold value according to the description in Section 3.2. In this work, we randomly select four test sequences, T_0 , T_1 , T_6 , T_7 , for learning the threshold value. For each of them, we run Verilog simulation and generate the graph of cumulative new transition counts, as shown in Figures 3.11-3.12. We obtain the critical points for T_0 , T_1 , T_6 , T_7 are 130th, 126th, 195th, and 195th clock cycle, respectively. We then record the aggregated TCs for all registers at the critical point for each of the four target test sequences, i.e., $(1116700 \ 1082340 \ 1675050 \ 1675050)$. Since there are altogether 12 registers in the Scheduler module, we set the threshold

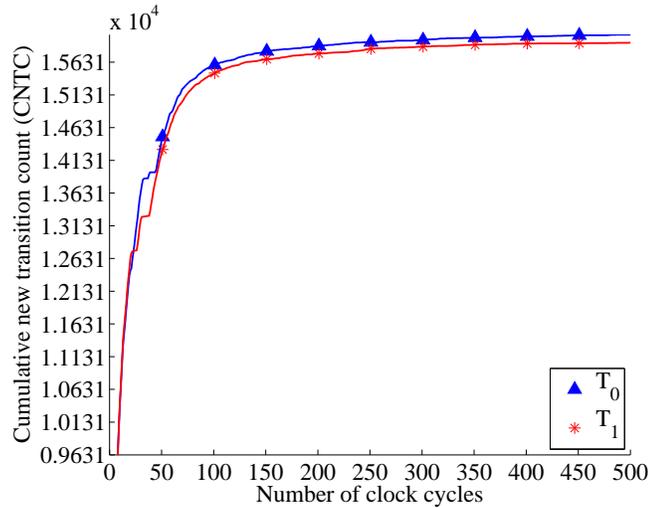


Figure 3.11: Cumulative new transition counts for T_0 and T_1 (Scheduler module).

value TH_1 to be $(1116700/12 + 1082340/12 + 1675050/12 + 1675050/12)/4$, i.e., 115,610. In the above setting of threshold value, the CNTC of the critical point is 98% of the maximum CNTC. We can also consider other percentage when obtaining critical point. For example, for a given percentage of 95%, we obtain the threshold value as 74,447, labeled as TH_2 . For a percentage of 90%, we obtain the threshold value as 43,129, labeled as TH_3 .

Given the information of the TCs, the weight vector, the observability vector and the threshold value, RT-level deviations can be calculated. The corresponding RT-level deviations for the Scheduler module are shown in Table 3.4. The total CPU time for deviation calculation and test-sequence grading is less than 8 hours. The CPU time for gate-level stuck-at (transition) fault simulation is 110 (175) hours, and the CPU time for computing the gate-level BCE (BCE^+) measure is 120 (125) hours, thus we are able to reduce CPU time by over an order of magnitude..

Correlation between output deviations and gate-level fault coverages

We obtain the stuck-at and transition fault coverages for the functional test sequences

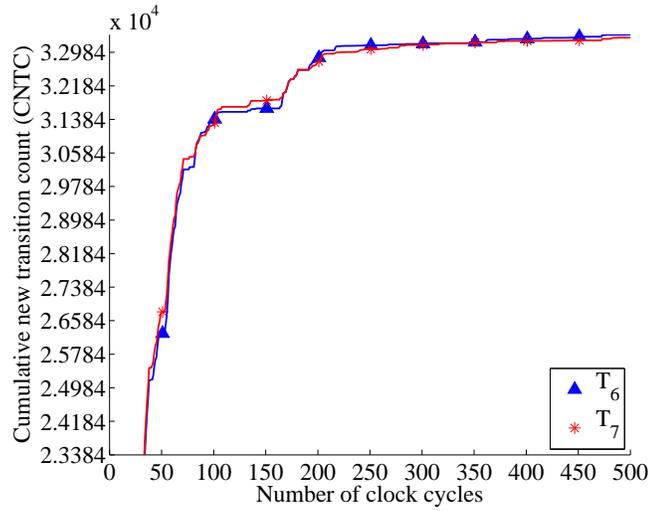


Figure 3.12: Cumulative new transition counts for T_6 and T_7 (Scheduler module).

by running fault simulation at the gate-level. Bridging fault coverage is estimated using the BCE and BCE^+ metrics. The correlation between these gate-level fault coverage measures and RT-level deviations are computed and the Kendall's correlation coefficients are shown in Figure 3.13. As in the case of the Biquad filter, the correlation coefficients are close to 1 for all the three threshold values. This demonstrates that the RT-level deviations are a good predictor of the gate-level fault coverage.

As in the case of the Biquad filter, if observability and weight vectors are ignored and only transition counts are considered in the calculation of output deviations, we observe a considerable degradation in the correlation coefficients between output deviations and gate-level fault coverage. These results are shown in Figure 3.14.

Cumulative gate-level fault coverage (Ramp-up) We evaluate the cumulative gate-level fault coverage of several reordered functional test sequences. Besides traditional stuck-at and transition fault coverage, we use BCE^+ metric to evaluate unmodeled defect coverage.

These reordered sequences sets are obtained in four ways: (i) baseline order

Table 3.4: RT-level deviations of ten functional tests for Scheduler module.

Threshold value			
Functional test	TH_1	TH_2	TH_3
T_0	0.3324	0.3424	0.3480
T_1	0.3324	0.3424	0.3476
T_2	0.3304	0.3400	0.3456
T_3	0.3308	0.3408	0.3464
T_4	0.3320	0.3420	0.3472
T_5	0.3324	0.3424	0.3476
T_6	0.4220	0.4324	0.4380
T_7	0.4256	0.4364	0.4420
T_8	0.3652	0.3772	0.3840
T_9	0.3680	0.3804	0.3868

T_0, T_1, \dots, T_9 ; (ii) random ordering $T_2, T_1, T_5, T_7, T_0, T_3, T_8, T_9, T_6, T_4$; (iii) random ordering $T_4, T_1, T_0, T_9, T_2, T_7, T_6, T_3, T_5, T_8$; (iv) the descending order of output deviations. In the deviation-based method, test sequences with higher deviations are ranked first. Figure 3.15-3.17 show cumulative stuck-at, transition fault coverages and $BCE+$ for the four reordered functional test sequences following the above four orders. We find that the deviation-based method results in the steepest cumulative stuck-at, transition and bridging fault coverage curves.

Comparison of proposed method with [39]

A method based on traversed circuit states was proposed recently to estimate the gate-level fault coverage of functional test sequences [39]. We consider [39] as a baseline and compare it to the results obtained by the proposed method.

Based on gate-level fault coverage, we can obtain an ordering of the given functional test sequences. We consider this order to be golden (desired) ordering. Based on the RT-level output deviations and the estimated gate-level fault coverage as in reference [39], we obtain two other orders of these sequences, labeled as *Order_dev*

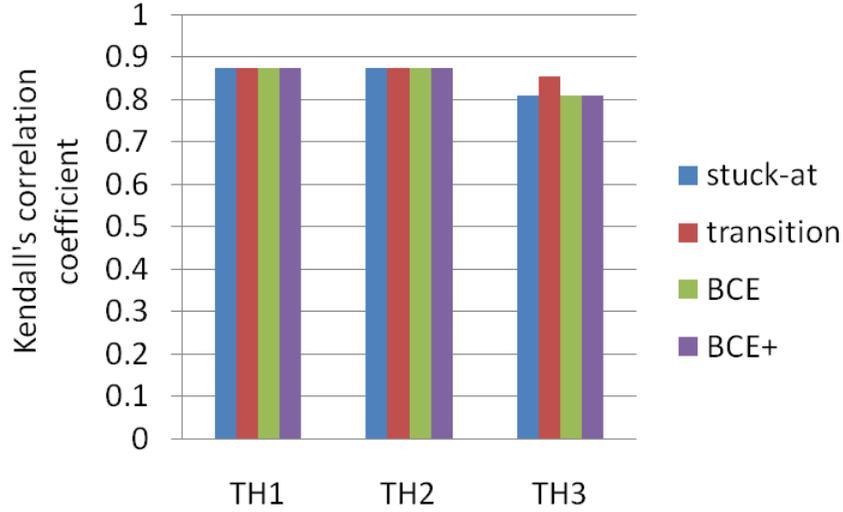


Figure 3.13: Correlation between output deviations and gate-level fault coverage (Scheduler module).

and *Order_Ref*[39]. Our goal is to observe whether *Order_dev* has more similarity with the golden order compared to *Order_Ref*[39]. The problem of comparing the similarity of two orders can be viewed as a sequence-matching problem. The

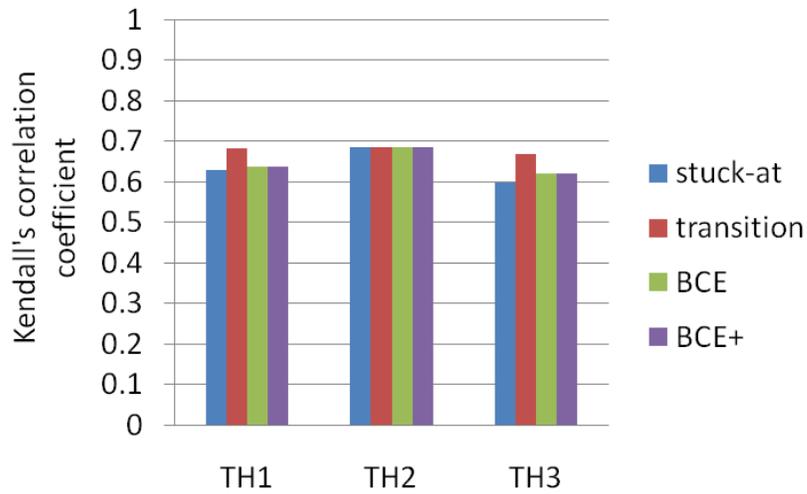


Figure 3.14: Correlation between output deviations (only consider transition counts) and gate-level fault coverage (Scheduler module).

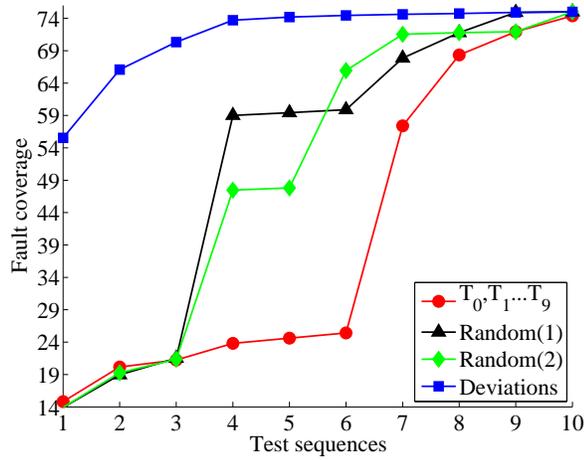


Figure 3.15: Cumulative stuck-at fault coverage for various test-sequence ordering methods (Scheduler module).

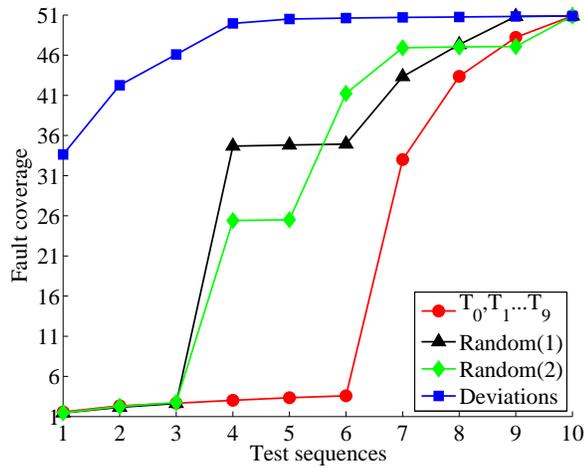


Figure 3.16: Cumulative transition fault coverage for various test-sequence ordering methods (Scheduler module).

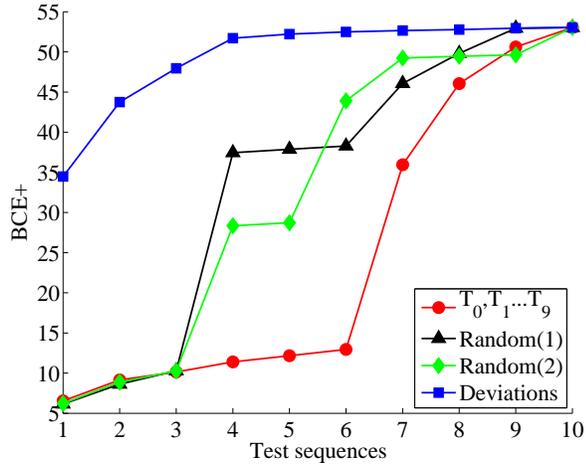


Figure 3.17: Cumulative $BCE+$ metric for various test-sequence ordering methods (Scheduler module).

sequence-matching problem has been extensively studied in the literature [79] [80]. We choose a widely used metric, namely the longest common subsequence (LCS), to evaluate the similarity of two sequences.

In our problem, we can simply calculate the common subsequence starting with every functional test sequence, and then choose the longest common subsequence as the LCS of the orders. For example, suppose two orders to be compared are $TS1 > TS2 > TS3 > TS4 > TS5$ and $TS2 > TS3 > TS4 > TS1 > TS5$. There are no common subsequences starting with $TS1, TS4$ and $TS5$. The common subsequence starting with $TS2$ is $TS2 > TS3 > TS4$; the common subsequence starting with $TS3$ is $TS3 > TS4$. Therefore, the LCS of these two orders is $TS2 > TS3 > TS4$, and the length of the LCS is 3.

Table 3.5 compares the proposed method to [39] based on the length of the LCS. We can see that the length of LCS for *Order_dev* and the golden order based on stuck-at fault coverage is 4, while the length of LCS for *Order_Ref*[39] and the golden order based on stuck-at fault coverage is 3. In other words, the ordering based on the proposed method has more similarity with the golden order. Similar

Table 3.5: Length of LCS.

Order	<i>Order_dev</i>	<i>Order_Ref</i> [39]
Stuck-at fault coverage	4	3
Transition fault coverage	4	2
<i>BCE+</i>	3	3

results are obtained when we consider the transition fault model. For the *BCE+* measure, the LCS length for the two methods are the same. This implies that the proposed method is more effective in grading functional test sequences than [39]. The reason may lie in the fact that observability of registers is considered in the proposed method but it is not considered in reference [39]. Since observability is an important factor that affects fault coverage, the proposed method can provide a more effective ordering of functional test sequences.

Another advantage of the proposed method is that it is scalable to large designs since it requires less computation time. The total CPU time for deviation calculation and test-sequence grading for the Scheduler module is less than 8 hours. On the other hand, the CPU time for grading test sequences based on [39] is about 73 hours. This is because the method in [39] requires logic simulation of the gate-level circuit. Therefore, [39] is not scalable for more realistic scenarios, where more functional test sequences have to be considered. Furthermore, with the proposed RT-level deviation based method, we are able to reduce CPU time by nearly a order of magnitude, with better or comparable test-sequence grading results.

Results for different gate-level implementations

There can be different gate-level implementations for the same RT-level design. A gate-level netlist is usually obtained by applying synthesis procedure to a given RT-level design. Different technology libraries or synthesis constraints such as timing

Table 3.6: Comparison of three implementations.

	<i>imp1</i>	<i>imp2</i>	<i>imp3</i>
Technology library	FREEPDK	FREEPDK	GSCLib3.0
Map_effort	medium	high	medium
No. of gates	375,061	300,422	296,367
No. of flip-flops	8,590	8,590	8,590
No. of stuck-at faults	1,459,262	974,298	1,113,422

and area requirements will lead to different gate-level netlists for the same RT-level design. For the Scheduler module, all the experiments reported thus far were carried out using a specific gate-level netlist, which we call *imp1*. We next evaluate whether the proposed RT-level deviation-based functional test grading method is sensitive to gate-level implementation. We first obtain two other gate-level netlists, namely *imp2* and *imp3*, for the given Scheduler module. Note that Design Compiler (DC) from Synopsys was used in synthesis procedure to obtain the gate-level netlists. Next, we re-calculate their RT-level output deviations and observe the correlations between deviations with gate-level fault coverage metrics.

Table 3.6 shows the comparison for these three implementations. Note that *imp1* and *imp2* adopt the FREEPDK library in synthesis while *imp3* uses the GSCLib3.0 library. FREEPDK is a 45 nm gate library provided by FreePDF and GSCLib3.0 is a 180 nm generic library from Cadence. With respect to the synthesis constraints, *imp2* is obtained by setting map_effort to be high, while *imp1* and *imp3* are obtained by setting map_effort to be medium. When the option map_effort is set high, the synthesis tool spends more effort on improving the performance. The total numbers of gates, flip-flops, and stuck-at faults for each implementation are also listed in Table 3.6.

RT-level output deviations are calculated based on transition counts, the weight vector, the observability vector and the threshold value. To calculate RT-level output

Table 3.7: RT-level deviations of ten functional tests for *imp2* of Scheduler module.

Threshold value			
Functional test	<i>TH_1</i>	<i>TH_2</i>	<i>TH_3</i>
T_0	0.4032	0.4248	0.4320
T_1	0.4032	0.4244	0.4320
T_2	0.4012	0.4224	0.4296
T_3	0.4016	0.4232	0.4304
T_4	0.4024	0.4240	0.4316
T_5	0.4028	0.4244	0.4320
T_6	0.4916	0.5144	0.5216
T_7	0.4960	0.5180	0.5252
T_8	0.4352	0.4592	0.4672
T_9	0.4376	0.4620	0.4704

deviations for *imp2* and *imp3*, we only need to re-calculate the weight vector since transition counts, observability vector and threshold value remains unchanged for the same functional test and the same RT-level design. The re-calculated RT-level deviations under three threshold value (same as for *imp1*) are shown in Table 3.7 and Table 3.8. As expected, the deviations for the three implementations are different.

After we have obtained the RT-level deviations for *imp2* and *imp3*, we calculate the gate-level fault coverage measures for them. Finally we compute the correlation between RT-level deviations and gate-level fault coverage measures. The Kendall's correlation coefficient values are shown in Figure 3.18 and Figure 3.19. As in the case of *imp1*, the correlation coefficients are close to 1 for all the CL vectors for *imp2* and *imp3*. These results highlight the fact that RT-level deviations are a good predictor of gate-level defect coverage for different gate-level implementations of an RT-level design. We therefore conclude that RT-level deviations can be used to grade functional test sequences for different gate-level implementations.

Table 3.8: RT-level deviations of ten functional tests for *imp3* of Scheduler module.

Functional test \ Threshold value	TH_1	TH_2	TH_3
T_0	0.4232	0.4388	0.4468
T_1	0.4232	0.4388	0.4468
T_2	0.4212	0.4364	0.4444
T_3	0.4216	0.4372	0.4452
T_4	0.4228	0.4384	0.4460
T_5	0.4232	0.4388	0.4464
T_6	0.5132	0.5284	0.5360
T_7	0.5164	0.5324	0.5400
T_8	0.4564	0.4732	0.4820
T_9	0.4588	0.4764	0.4848

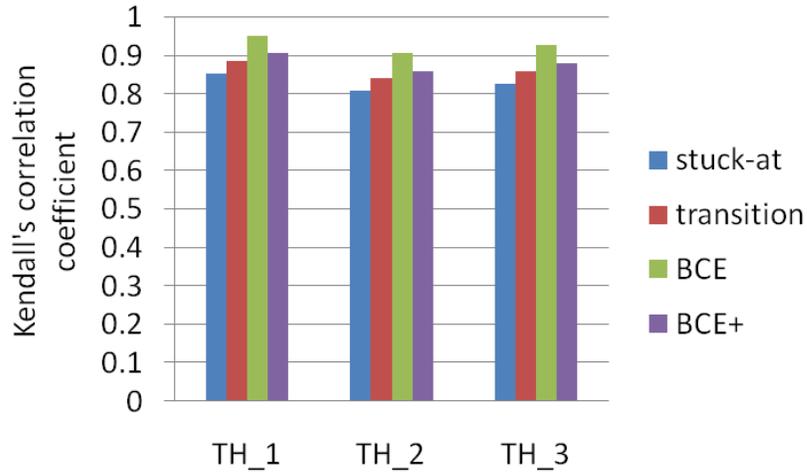


Figure 3.18: Correlation between output deviations and gate-level fault coverage for *imp2* (Scheduler module).

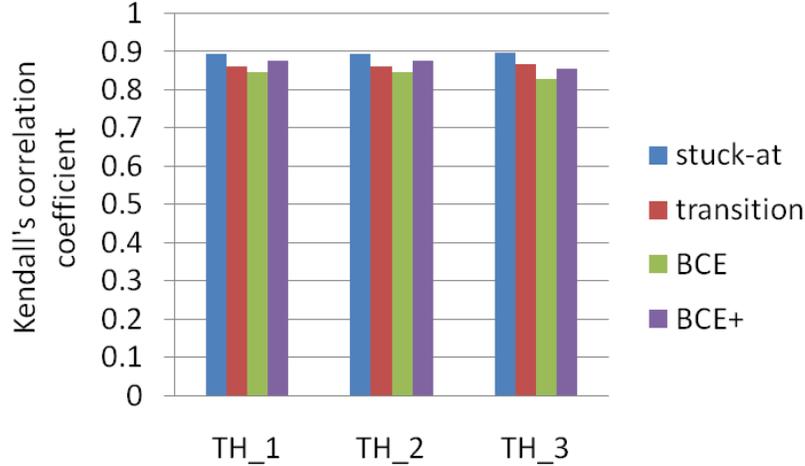


Figure 3.19: Correlation between output deviations and gate-level fault coverage for *imp3* (Scheduler module).

Stratified sampling for threshold-value setting

The threshold value is a key parameter in the calculation of RT-level output deviations. We propose to set the threshold value for a design by learning from a subset of the given functional test sequences. However, if we randomly select several test sequences as the learning sequences, they may not represent the characteristics of the entire test set and thus may lead to inaccurate threshold values.

Here we address the problem of threshold-value setting from a survey sampling perspective [81]. The given N functional test sequences constitute the population for the survey. The $CNTC$ is the characteristic under study. The threshold value is estimated by simulating the design using a number of samples drawn from the population, a procedure referred to as sampling. Our objective is to design a sampling procedure that will significantly reduce the number of simulated test sequences while ensure the accuracy of the estimated threshold value.

Stratified sampling [81] has been widely used for surveys because of its efficiency. The purpose of stratification is to partition the population into disjoint subpopula-

Table 3.9: Correlation results with stratified sampling (Scheduler module).

Sampling case	Sample size of <i>Group1</i>	Sample size of <i>Group2</i>	Kendall's Correlation Coefficient			
			<i>Stuck_at</i>	<i>Transition</i>	<i>BCE</i>	<i>BCE+</i>
1	1	1	0.8540	0.8989	0.8540	0.8540
2	2	2	0.8741	0.8741	0.8741	0.8741
3	2	2	0.8667	0.8819	0.8540	0.8667
4	3	2	0.8667	0.9111	0.8667	0.8741
5	6	4	0.9080	0.8819	0.8810	0.8810

tions, called strata, so that the characteristic of members within each subpopulation is more homogeneous than in the original population. In this paper, we partition the given functional test sequences into subgroups according to their sources. For the Scheduler module, we partition its 10 functional test sequences into two subgroups: *Group1* composed of 6 functional test sequences coming from Intel and *Group2* composed of 4 functional test sequences from ATPG. (To apply the stratified sampling to general designs, we may need further information about functional test sequences, such as the functionality of each test or the targeting modules of each test, to partition the given functional test sequences into appropriate subgroups.)

If the sample is taken randomly from each stratum, the procedure is known as stratified random sampling. We use stratified random sampling for the Scheduler module. For a given sample size, we first adopt random sampling for *Group1* and *Group2*. Next we run simulation on the sampled functional test sequences and obtain the threshold value. Then based on this threshold value, we calculate the RT-level output deviations. Finally we compute the Kendall's correlation coefficient of deviation metric and gate-level fault coverage.

Table 3.9 shows the correlation results for the Scheduler module when different stratified random sampling choices are used. The first column list the different sampling. Column 2 and Column 3 list the sample size for each subgroup. For example,

for Sample 4, we sampled 3 functional test sequences from *Group1* and 2 functional test sequences from *Group2*. Columns 4 – 7 list the correlation coefficient of RT-level output deviations (calculated based on the sampling) and the gate-level fault coverage. We can see that by setting threshold value using stratified sampling in the calculation of RT-level deviations, we consistently obtain good correlation results. This demonstrates the effectiveness of stratified sampling in setting threshold values. Also, we observe that in most cases, the correlation coefficient increases with the sample size. Therefore, the sample size m can be chosen based on how much pre-processing can be carried out in a limited amount of time.

Results for more functional test sequences

To further verify the effectiveness of the RT-level output deviation metric in grading functional test sequences, we consider more functional test sequences for the Scheduler module. For the test sequences generated by sequential ATPG, we performed 3 bit-operations (“not”, “xnor” and “xor”) to obtain 3 new functional test sequences. Similarly, for each of the six functional test sequences provided by Intel, we perform the same bit-operations and obtain 3 new functional test sequences. In this way, we obtain 28 functional test sequences altogether.

We calculate the RT-level output deviations and record the gate-level fault coverage for these 28 functional test sequences. Then we compute the Kendall’s correlation coefficient as in the case of 10 functional test sequences. The correlation results are shown in Figure 3.20. We can see that the correlation coefficient is very close to 1. Compared to the case of 10 functional test sequences (Figure 3.13), the correlation is higher. This implies that our proposed method is more effective for grading a larger number of functional test sequences.

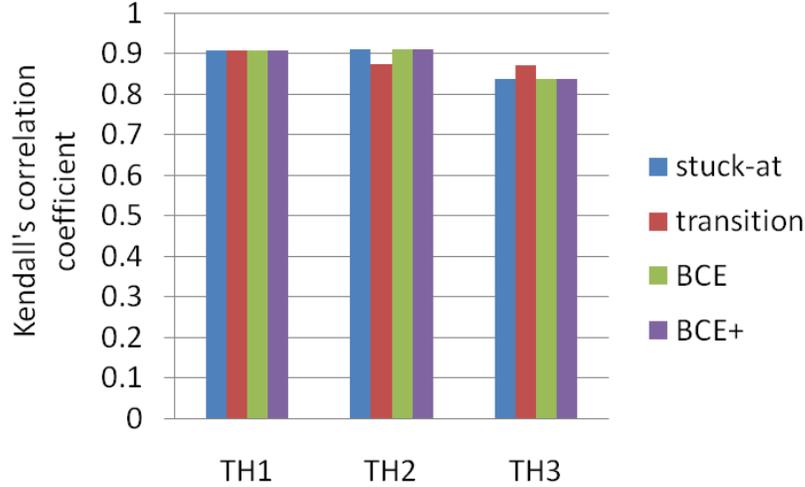


Figure 3.20: Correlation results for more functional test sequences (Scheduler module).

3.4 RTL DFT for Functional Test Sequences

Functional test sequences often suffer from low defect coverage since they are mostly derived in practice from existing design-verification test sequences. Therefore, there is a need to increase their effectiveness using design-for-testability (DFT) techniques. We present a DFT method that uses the register-transfer level (RTL) deviations metric to select observation points for an RTL design and a given functional test sequences. Simulation results for six *ITC'99* circuits show that the proposed method outperforms two baseline methods for several gate-level coverage metrics, including stuck-at, transition, bridging, and gate-equivalent fault coverage. Moreover, by inserting a small subset of all possible observation points using the proposed method, significant fault coverage increase is obtained for all benchmark circuits.

3.4.1 Problem Formulation

We first formulate the problem being tackled in this chapter.

Given:

- RT-level description for a design and a functional test sequence \mathcal{S} ;
- A practical upper limit n on the number of observation points that can be added.

Goal: Determine the best set of n observation points that maximizes the effectiveness of the functional test sequence \mathcal{S} .

The functional test sequences are typically derived manually and used for design verification, or semi-automatically generated by RT-level test generation methods. These instructions can be instruction-based for processors or application-based for application specific integrated circuit (ASIC) cores such as an MPEG decoder. They can be in the format of high-level instructions or commands, or in the format of binary bit streams.

To increase testability, we can insert an observation point for each register output. We can obtain the highest defect coverage by inserting the maximum number of observation points. However, it is impractical to do so due to the associated hardware and timing overhead. In fact, the number of observation points that can be added is limited in practice. For a given upper limit n , the challenge is to determine the best set of n observation points such that we can maximize the defect coverage of the given functional test sequence. Our main premise is that RT-level output deviation can be used as a metric to guide observation-point selection.

3.4.2 Observation-Point Selection

In this section, we first define the new concept of RT-level internal deviations. Next, we analyze the factors that determine observation-point selection. Then we present the observation-point selection algorithm based on RT-level deviations. Finally, we introduce recent related work [32], which will be used in Section 3.4.3 for comparison.

RT-Level Internal Deviations

The RT-level output deviation [62] is defined to be a measure of the likelihood that error is manifested at a primary output. Here we define the RT-level internal deviation to be a measure of the likelihood of error being manifested at an internal register node, which means error being manifested at one or more bits of register outputs. In the calculation of RT-level output deviation, a transition in a register is meaningful only when it is propagated to a primary output. On the other hand, in the calculation of RT-level internal deviation, we do not care whether a transition in a register is propagated to a primary output. The method for calculating internal deviations for register can also be used to calculate internal deviations for each bit of a register.

The calculation of RT-level internal deviation is similar to that of RT-level output deviation. In order to calculate RT-level internal deviations for a register, first we need to define the internal effective TCs. Suppose a register Reg makes N_1 0→1 transitions for a functional test sequence TS . Then its internal effective 0→1 TCs equals the product of N_1 and the weight of register Reg . The observability value of register Reg does not contribute to its effective transition count. Take IR in the Parwan processor as an example. It has the observability value 0.5 and weight 0.1556. Since it makes 67 0→1 transitions for functional test sequence TS , its internal effective 0→1 transition count is 0.1556×67 , i.e., 10.43. In the same way, we can calculate the internal effective TCs of all registers for TS , as shown in Table 3.10.

After obtaining the internal effective transition counts, we can calculate the RT-level internal deviations. Suppose a functional test sequence TS is composed of m instructions I_1, I_2, \dots, I_m . For each instruction I_i , suppose the internal effective 0→0 TCs for register R_k ($1 \leq k \leq t$) is $I-R_{ki00}$, the internal effective 0→1 TCs for register R_k is $I-R_{ki01}$, the internal effective 1→0 TCs for register R_k is $I-R_{ki10}$, and

Table 3.10: Internal effective TCs for test sequence TS .

Register ID	Register name	0→0	0→1	1→0	1→1
R_1	AC	67	61	60	44
R_2	IR	32.06	10.43	10.26	5.76
R_3	PC	305.08	38.78	36.63	88.33
R_4	MAR	84.91	23.34	22.88	22.88
R_5	SR	62.32	10.23	13.02	11.16

the internal effective 1→1 TCs for register R_k is $I_{-}R_{ki11}$. Let $I_{-}S_{k00} = \sum_{i=1}^m I_{-}R_{ki00}$, $I_{-}S_{k01} = \sum_{i=1}^m I_{-}R_{ki01}$, $I_{-}S_{k10} = \sum_{i=1}^m I_{-}R_{ki10}$, $I_{-}S_{k11} = \sum_{i=1}^m I_{-}R_{ki11}$. The internal deviation of register R_k for TS can be calculated for a given CL vector $(cl_{00}, cl_{01}, cl_{10}, cl_{11})$ as follows:

$$I_{dev}(R_k) = 1 - cl_{00}^{I_{-}S_{k00}} \cdot cl_{01}^{I_{-}S_{k01}} \cdot cl_{10}^{I_{-}S_{k10}} \cdot cl_{11}^{I_{-}S_{k11}}. \quad (3.5)$$

Note that $I_{-}S_{k00}$ is the aggregated internal effective 0→0 TCs of register R_k for all the instructions in TS . The parameters $I_{-}S_{k01}$, $I_{-}S_{k10}$, and $I_{-}S_{k11}$ are defined in a similar way. In this way, we can calculate the internal deviations of every register for TS . The method for calculating internal deviations for register can also be used to calculate internal deviations for each bit of a register.

Analysis of Factors that Determine Observation-Point Selection

The selection of observation points is determined by three factors: RT-level internal deviations of registers, observability values of registers, and the topological relationship between registers. In this work, we only consider the insertion of observation points at outputs of registers.

For a register Reg , we have the following attributes attached with it: $I_{dev}(Reg)$, $O_{dev}(Reg)$, $obs(Reg)$, to represent its internal deviation, output deviation, and observability value, separately.

For two registers $Reg1$ and $Reg2$, when two attributes are close in value, we define the following observation-point-selection rules based on the third attribute:

Rule 1: If $I_{dev}(Reg1) > I_{dev}(Reg2)$, select $Reg1$;

Rule 2: If $obs(Reg1) < obs(Reg2)$, select $Reg1$;

Rule 3: If $Reg1$ is the logical predecessor of $Reg2$, select $Reg2$.

For Rule 1, the motivation is that if we select a register with higher I_{dev} , its observability will become 1. Thus, its O_{dev} will also become higher. The higher O_{dev} of this register will contribute more to the cumulative O_{dev} for the circuit. Since we have shown that the cumulative O_{dev} is a good surrogate metric for gate-level fault coverage [62], we expect to obtain better gate-level fault coverage when we select a register with higher I_{dev} .

For Rule 2, when two registers do not have a predecessor/successor relationship with each other, obviously we should select the register with lower observability. For Rule 3, if we select $Reg1$, $obs(Reg1)$ will become 1 but this will not contribute to the increase of observability of $Reg2$; if we select $Reg2$, $obs(Reg2)$ will become 1 and $obs(Reg1)$ will also be increased due to the predecessor relationship between $Reg1$ and $Reg2$. Therefore, it is possible that the selection of $Reg2$ yields better results than the selection of $Reg1$, i.e., the cumulative observability after the insertion of observation point on $Reg2$ is higher than for $Reg1$.

Rule 2 and Rule 3 are in conflict with each other on the observability attribute. Rule 2 selects a register with lower observability while Rule 3 selects a register with higher observability. In this work, we assume that Rule 3 is given higher priority than Rule 2.

We use RT-level deviations to guide the selection of observation points. We have determined that we should select a register with higher I_{dev} . Since O_{dev} is proportional to I_{dev} and obs , if I_{dev} factor contributes more to O_{dev} , we should select

the register with higher O_{dev} . Also, by selecting a register with higher O_{dev} , we are implicitly satisfying the predecessor relationship rule: for two registers $Reg1$ and $Reg2$ whose I_{dev} values are comparable, if $Reg1$ is the predecessor of $Reg2$, we have $obs(Reg1) < obs(Reg2)$ and $O_{dev}(Reg1) < O_{dev}(Reg2)$. Then we will not select $Reg1$, which is in accordance with Rule 3.

RT-Level Deviation Based Observation-Point Selection

Based on the RT-level deviations, we have developed a method for selecting best set of n (where n is a user-specified parameter) observation points for a given RT-level design and a given functional test sequence. In the selecting of observation-points, we target the specific bits of a register. The calculation of I_{dev} , O_{dev} , obs for a register is carried out for each bit of a register. The selection procedure is as following:

- Step 0: Set the candidate list to be all bits of registers that do not directly drive a primary output.
- Step 1: Derive the topology information for the design and save this information in a look-up table. Obtain the weight vector, observability vector, and TCs for each register bit, and calculate RT-level output deviations for each register bit.
- Step 2: Select a register bit with the highest output deviations as an observation point. Remove this selected register bit from the candidate list.
- Step 3: If the number of selected observation points reaches n , terminate the selection procedure.
- Step 4: Update the observability vector using the inserted observation point (selected in Step 2) and the topology information. Re-calculate output deviations for each register bit using the updated observability vector. Go to Step 2.

In Step 1, the topology information of the design can be extracted using a design analysis tool, e.g., Design Compiler from Synopsys. It only needs to be determined once and it can be saved in a look-up table for subsequent use. In Step 4, after selecting and inserting an observation point, we need to update the observability vector because the observability of its upstream nodes will also be enhanced. There is no need to recompute TCs since these depend only on the functional test sequence, and they are not affected by the observation points. There is also no need to recalculate the weight vector.

After the n observation points have been selected, they are inserted in the original RT-level design. The modified RTL design is synthesized to a gate-level netlist. To insert an observation point, we simply need to connect it directly to a new primary output. An alternative method is to use only one additional primary output and connect all observation points to this primary output through *XOR* gates (space compactor). By doing so, we can reduce the number of extra primary outputs to one. However, this method will lead to lower fault coverage due to error masking.

Observation-Point Selection Based on Probabilistic Observability Analysis

An automatic method to select internal observation signals for design verification was proposed in recent work [32]. Since this method is also applicable for observation-point selection in manufacturing test, we take it as an example of recent related work and compare the results obtained by the proposed RT-level deviation based method to this method in Section 3.4.3.

3.4.3 Experimental Evaluation

We evaluated the efficiency of the proposed RT-level observation-point selection method by performing experiments on six *ITC'99* [35] circuits. These circuits are

translated into Verilog format and are taken as the experimental vehicles. The functional test sequences are generated using the RT-level test generation method from [35]. Our goal is to show that the RT-level deviation-based observation-point selection method can provide higher defect coverage than other baseline methods. Here, defect coverage is estimated in terms of following gate level coverage metrics:

- stuck-at fault coverage;
- transition fault coverage;
- enhanced bridging fault coverage estimate (*BCE+*);
- gate-exhaustive (GE) score (GE score is defined as the number of observed input combinations of gates) [65] [82].

Since functional test sequences are usually used to target unmodeled defects that are not detected by structural test, we considered metrics *BCE+* and GE score, which are more effective for defect coverage, comparing to traditional stuck-at fault coverage and transition fault coverage. The GE score is defined as the number of the observed input combinations of gates. Here, “observed” implies that the gate output is sensitized to at least one of the primary outputs. We first compare the gate-level fault coverage for the original design to the design with all observation points inserted. Next we show the gate-level fault coverage for different observation-point selection methods.

Experimental Setup

All experiments were performed on a 64-bit Linux server with 4 GB memory. Synopsys Verilog Compiler (VCS) was used to run Verilog simulation and compute the deviations. The Flextest tool was used to run gate-level fault simulation. Design

Compiler (DC) from Synopsys was used to synthesize the RT-level descriptions as gate-level netlists and extract the gate-level information for calculating the weight vector. For synthesis, we used the library for Cadence 180nm technology. All other programs were implemented in C++ codes or Perl scripts.

Comparison of Gate-Level Fault Coverage for the Original Design to the Design with All Observation Points Inserted

Table 3.11 compares the gate-level fault coverage (stuck-at, transition and *BCE+*) for the original design to the design with all observation points inserted. The parameters *SFC%*, *TFC%*, *BCE+%* indicate the gate-level fault coverage for stuck-at faults, transition delay faults and bridging fault estimate, respectively. *#OP* lists the number of observation points. Table 3.12 compares the gate-level GE score for the original design to the design with all observation points inserted.

From these two tables, we can see that the gate-level fault coverage is not very high even when all observation points are inserted. There are two possible reasons for this: one reason is that the design suffers from low controllability. The other reason is that the quality of the given functional test sequences are not so effective for modeled fault. We can increase the gate-level fault coverage by improving the quality of functional test sequences or by inserting control points to the design. However, we focus here only on selection of observation points so that the given functional test sequences can be made more useful for manufacturing test. Therefore, it is of interest to determine the maximum gate-level fault coverage when all possible observation points are inserted, and to normalize the fault coverage to this maximum value when we evaluate the impact of inserting a subset of all possible observation points.

Table 3.11: Gate-level fault coverage (stuck, transition and $BCE+$) of the design before and after inserting all observation points.

Circuit	Original design			Design with all observation points			
	SFC%	TFC%	BCE+%	#OP	SFC%	TFC%	BCE+%
<i>b09</i>	59.18	47.93	45.58	27	82.8	67.86	70.13
<i>b10</i>	36.89	20.19	28.04	14	69.03	45.67	55.07
<i>b12</i>	50.25	26.67	29.91	115	55.23	31.92	33.52
<i>b13</i>	35.9	23.33	23.11	43	70.83	44.02	47.12
<i>b14</i>	83.95	74.6	74.52	161	92.34	83.32	81.23
<i>b15</i>	9.91	5.35	4.4	347	23.29	11.36	10.63

Table 3.12: Gate-level GE score of the design before and after inserting all observation points.

Circuit	Original design	Design with all observation points	
	GE score	#OP	GE score
<i>b09</i>	121	27	173
<i>b10</i>	132	14	330
<i>b12</i>	889	115	1005
<i>b13</i>	257	43	483
<i>b14</i>	8601	161	8934
<i>b15</i>	806	347	1987

Comparison of Normalized Gate-Level Fault Coverage for Different Observation-Point Selection Methods

By considering the fault coverage of a design with all observation points inserted to be 100%, we normalize the fault coverage of designs with a smaller number of observation points. Similarly, the normalized GE score is obtained by taking the GE score of a design with all observation points inserted as the reference. In this section, we compare the normalized gate-level fault coverage and normalized GE score for different observation-point selection methods.

For each circuit, we select the same number of n (for various values of n) observation points using different methods. Results for normalized gate-level fault coverage and normalized GE score are shown in the Figure 3.21-3.24. We compare the proposed method to [32] and to a baseline random observation-point insertion method.

The results show that the proposed method outperforms the two baseline methods for all six circuits. By inserting a small fraction of all possible observation points using the proposed method, significant increase in fault coverage and GE score are obtained for all circuits. For each circuit, it only costs several seconds to calculate RT-level deviations and select observation points. These results highlight the effectiveness of the RT-level, deviation-based observation-point selection method.

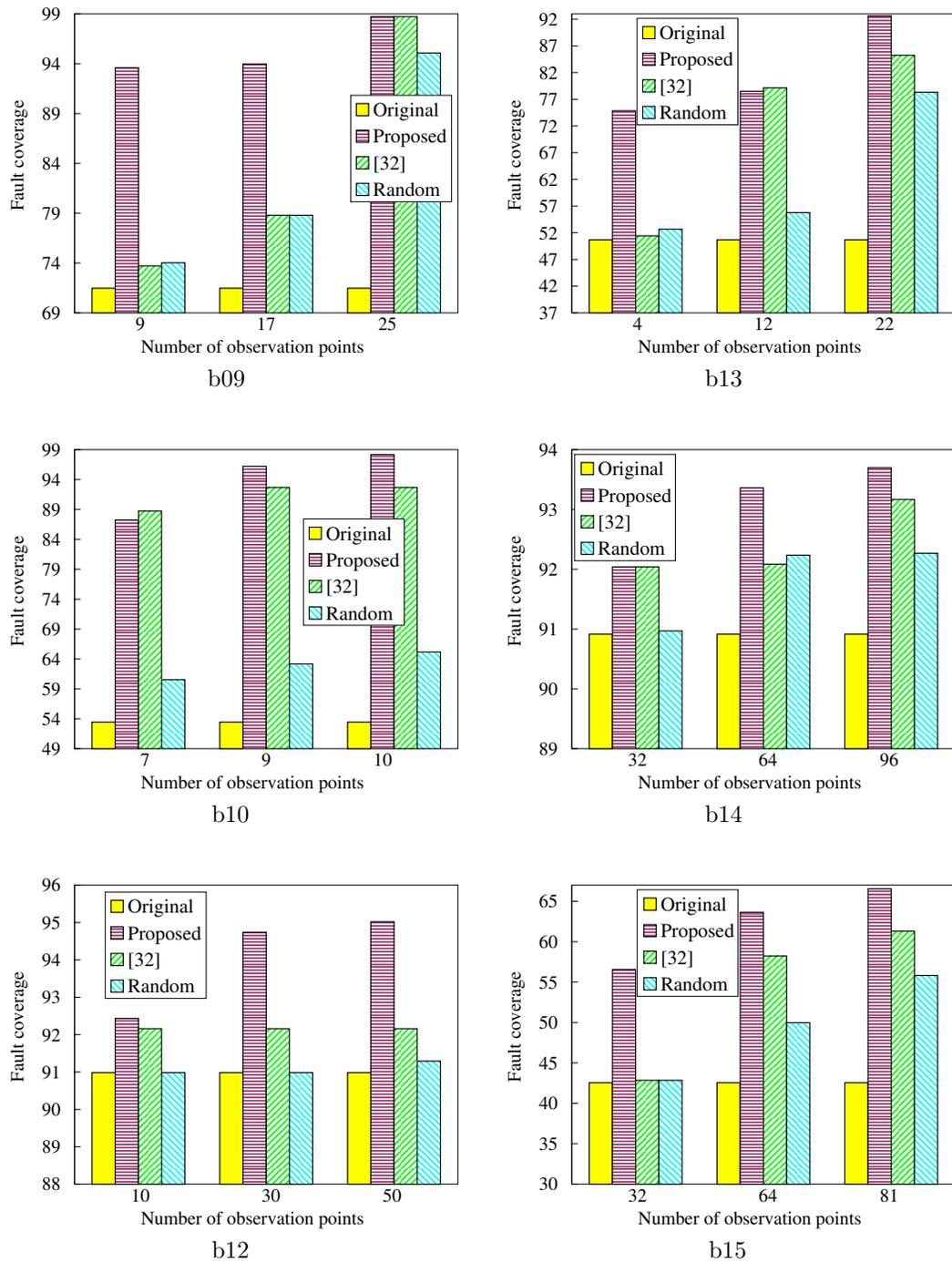


Figure 3.21: Results on gate-level normalized stuck-at fault coverage.

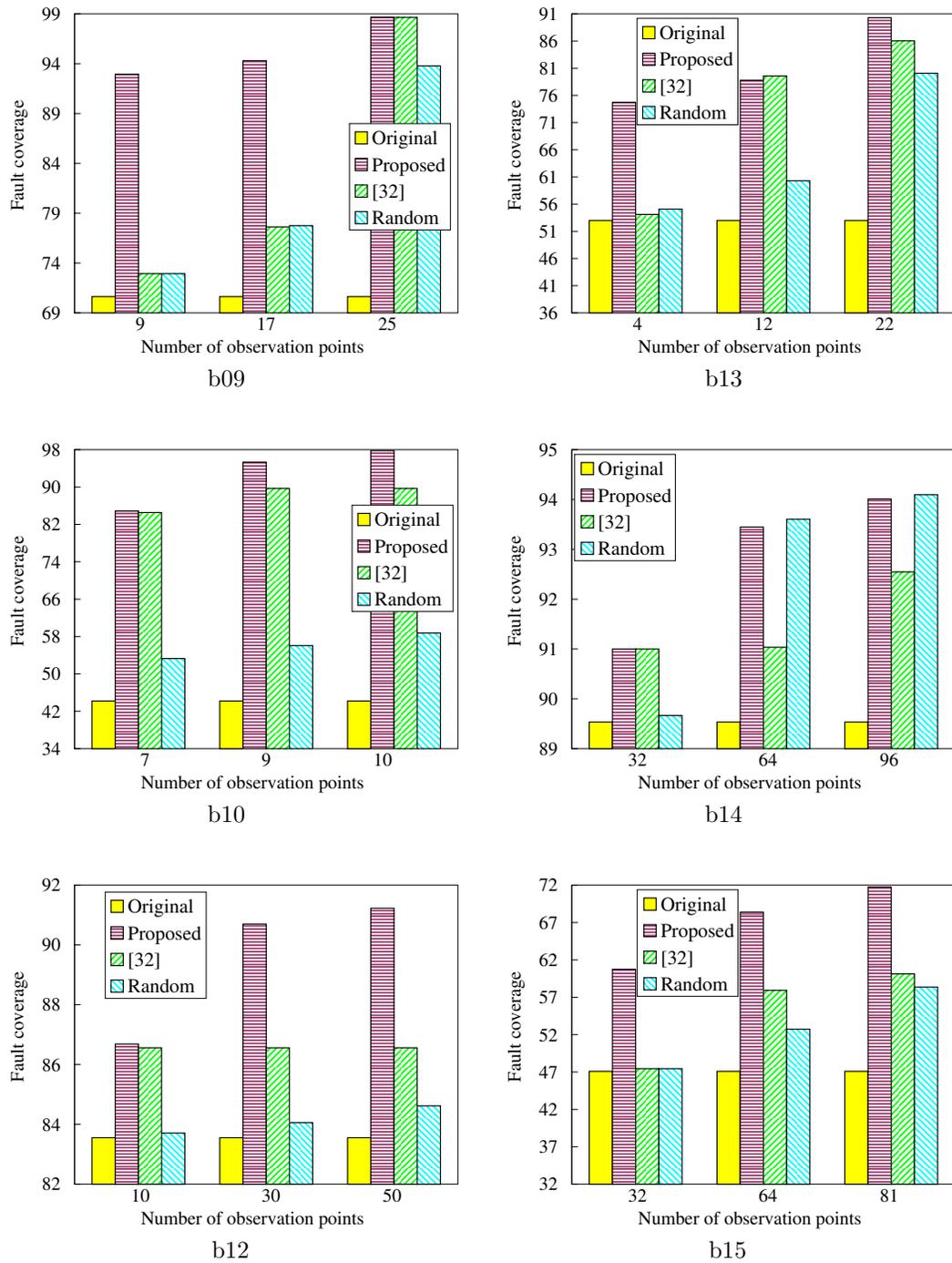


Figure 3.22: Results on gate-level normalized transition fault coverage.

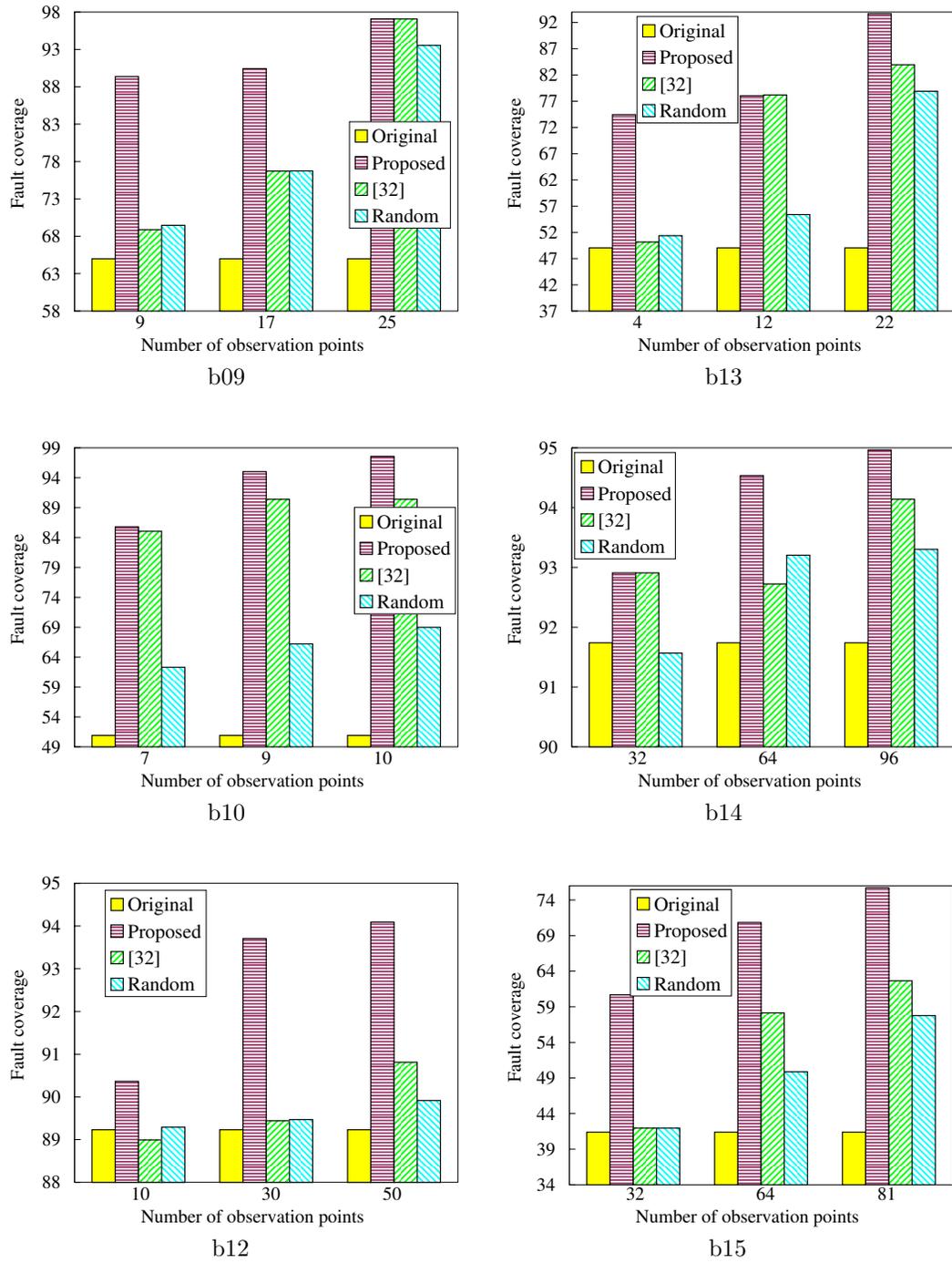


Figure 3.23: Results on the gate-level normalized $BCE+$ metric.

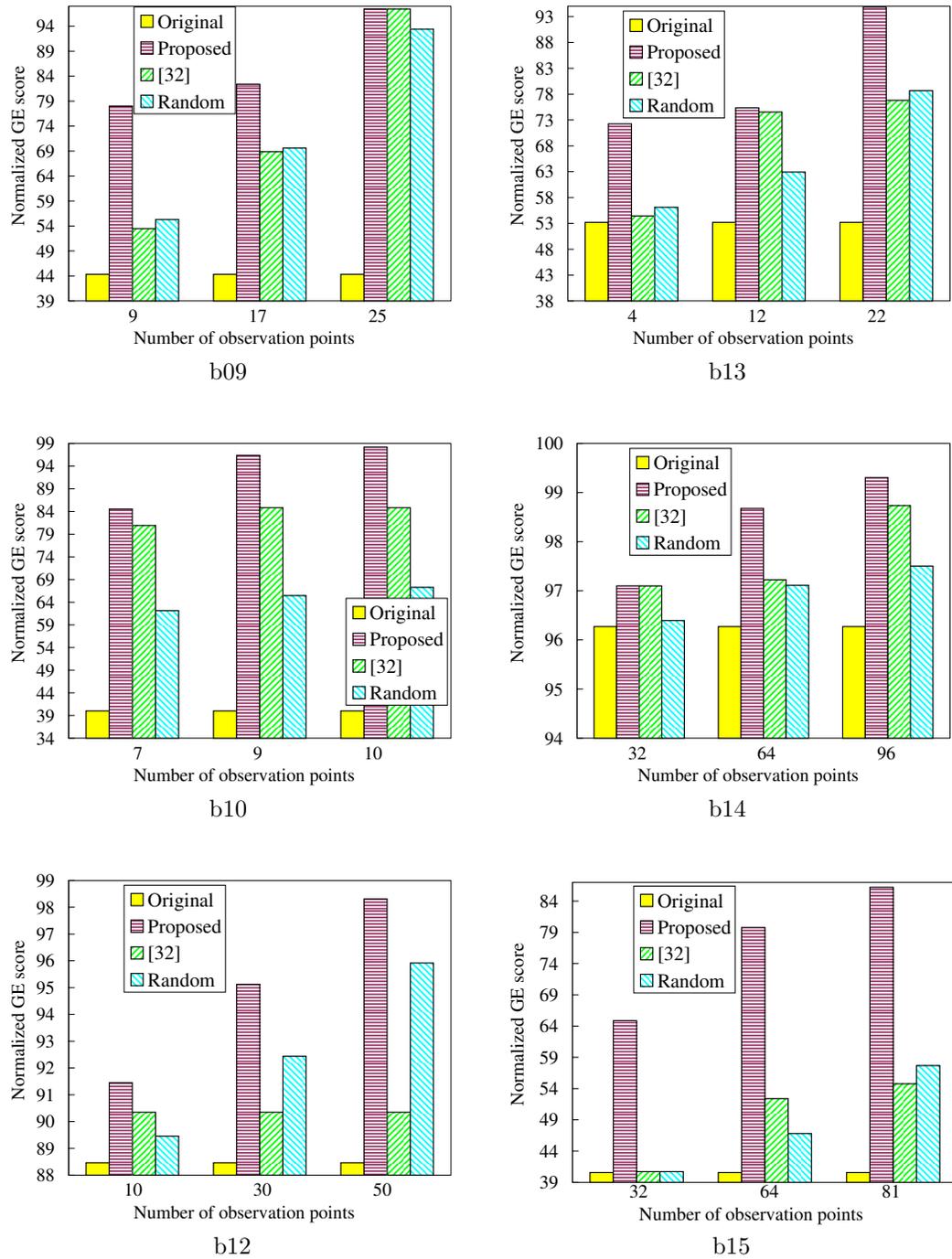


Figure 3.24: Results on gate-level normalized GE score.

3.5 Chapter Summary and Conclusions

In this chapter, we have presented the output deviation metric at RT-level to model the quality of functional test sequences. By adopting the deviation metric, timing-consuming fault simulation at gate-level can be avoided for the grading of functional test sequences. Experiments on the Biquad filter and the Scheduler module of the IVM design show that the deviations obtained at RT-level correlate well with the stuck-at, transition, and bridging fault coverage at the gate level. Moreover, the functional test sequences set reordered using deviations provides a steeper cumulative stuck-at and transition fault coverage as well as *BCE+* metric, and there is an order of magnitude reduction in CPU time compared to gate-level methods. We have also shown how the RT-level deviation metric can be used to select and insert the observation points for a given RT-level design and a functional test sequence. This DFT approach allows us to increase the effectiveness of functional test sequences (derived for pre-silicon validation) for manufacturing testing. Experiments on six *ITC'99* benchmark circuits show that the proposed RT-level DFT method outperforms two baseline methods for enhancing defect coverage. We also show that the RT-level deviations metric allows us to select a small set of the most effective observation points.

Chapter 4

Ranking Candidate Components/Blocks for Diagnosis of Board-Level Functional Failures

Despite recent advances in structural test methods, the diagnosis of the root cause of board-level failures for functional tests remains a major challenge. A promising approach to address this problem is to carry out fault diagnosis in two phases—suspect faulty components on the board or modules within components (together referred to as blocks in this chapter) are first identified and ranked, and then fine-grained diagnosis is used to target the suspect blocks in ranked order.

In this chapter, we propose a new method based on dataflow analysis and Dempster-Shafer theory for ranking faulty blocks in the first phase of diagnosis. The proposed approach transforms the information derived from one functional test failure into multiple-stage failures by partitioning the given functional test into multiple stages. A measure of “belief” is then assigned to each block based on the knowledge of each failing stage, and Dempster-Shafer theory is subsequently used to aggregate the beliefs from multiple failing stages. Blocks with higher beliefs are ranked at the top of the candidate list. Simulations on an industry design for a network interface application show that the proposed method can provide accurate ranking for most board-level functional failures. The proposed approach is computationally efficient since it avoids the need for time-consuming fault simulation procedures.

4.1 Dempster-Shafer Theory and Application

DS theory [93] was developed to address the drawbacks of traditional probability theory when it is applied to the conditions of epistemic uncertainty, i.e., the type of uncertainty that results from the lack of knowledge about a system. Traditional probabilistic analysis requires that an analyst have information about the probability of all events. For example, suppose a board is composed of 3 blocks: *block1*, *block2*, and *block3*. Suppose that when a board-level functional failure occurs, an expert assigns a probability of 0.4 for *block1* to be the root cause for this failure, and suppose that this expert has no basis to assign probabilities to the other two blocks being the root cause. Traditional probabilistic analysis is unable to capture this epistemic uncertainty and it might arbitrarily assign probabilities of 0.3 each to the two events corresponding to *block2* and *block3*. In DS theory, we are not forced to resort to assumptions about the probabilities of the individual events about which we are ignorant, i.e., we are not required to assign root-cause probabilities to *block2* and *block3*.

DS theory focuses on “belief” that an event will occur (or has occurred) rather than on the probability that it will occur. While traditional probability theory takes it as given that something is either true or false, DS theory allows for more nebulous states of a system, such as “unknown”. For example, suppose an expert believes that a board-level functional failure may be attributed to a root cause in *block1* with a likelihood of 0.3. In traditional probability theory, an equivalent way to present this statement is as follows: the expert believes that the root cause is not in *block1* with a likelihood of 0.7. However, without enough objective information, the expert’s subjective belief will be involved in the determination of the likelihood, which is not necessarily 0.7. DS theory can be exploited to manipulate the (subjective) belief. Returning to our example, an expert may have a belief of 0.3 for *block1* to be the

root cause and a belief of 0 for *block1* not to be the root cause. Here “0” implies that the expert has no evidence to support the statement that *block1* is not the root cause.

Moreover, DS theory allows us to combine evidence from different sources and arrive at a degree of belief (represented by a belief function) that takes into account all the available evidence. For example, suppose an expert states that *block1* is the root cause of a board-level functional failure. Suppose the probability that he is reliable is 0.9 and the probability that he is unreliable is 0.1. Thus the expert’s testimony alone justifies a 0.9 degree of belief that *block1* is the root cause, but only a zero degree of belief that *block1* is not the root cause. This zero merely means that the expert’s testimony gives no reason to believe that *block1* is not the root cause. Suppose another expert also states that *block1* is the root cause of a board-level functional failure and we have a 0.8 probability that he is reliable. The event that the first expert is reliable is independent of the event that the second expert is reliable, therefore, the probability that both experts are reliable is $0.9 \times 0.8 = 0.72$, the probability that neither is reliable is $0.1 \times 0.2 = 0.02$, and the probability that at least one is reliable is $1 - 0.02 = 0.98$. Both experts state that *block1* is the root cause, and if we assume that at least one of them is reliable, we can assign the event that *block1* is the root cause a degree of belief of 0.98.

In addressing the problem of board-level functional failure diagnosis, we cannot obtain an accurate *a priori* probability for the event that a given block is the root cause. Moreover, there is a possibility that a judgement cannot be made due to ignorance or lack of evidence. Therefore, DS theory emerges as a natural choice as a reasoning method. Furthermore, if a failure occurs when a given functional test is applied, we can partition this functional test into several stages and obtain the pass/fail information for each stage. Each failing stage can be viewed as one piece of evidence

from one source. These pieces are independent since the pass/fail information for each stage is collected by a separate simulation run based on only this stage. In this case, DS theory can be used to combine pieces of evidence from multiple sources, i.e., the multiple failing functional test stages, to arrive at a final determination about the probability for each block to be the root cause of the functional failure.

Consider a functional failure for a board composed of m blocks A_1, A_2, \dots, A_m . In this work, we assume that the root cause of a functional failure resides in only one block. Let $X = \{\xi_1, \xi_2, \dots, \xi_m\}$ be a set of events, where ξ_i represents the event that A_i is the root cause. We further use ϕ to denote the event that a judgement cannot be made. Let $\bar{X} = X \cup \{\phi\}$. According to DS theory, a belief mass is assigned to each element of \bar{X} based on one piece of evidence (one failing functional test stage). In particular, it is defined as a function $m_1 : \bar{X} \rightarrow [0, 1]$ with the property $\sum_{\xi_i \in X} m_1(\xi_i) + m_1(\phi) = 1$. Note that $m_1(\xi_i)$ is the belief for A_i to be the root cause, and $m_1(\phi)$ is the reserved belief, representing the degree of doubt regarding this evidence. Similarly, suppose additional evidence results in another belief function, i.e., $m_2 : \bar{X} \rightarrow [0, 1]$, with the property $\sum_{\xi_j \in X} m_2(\xi_j) + m_2(\phi) = 1$. Note that the two belief functions may assign mass to a possibly different set of elements. These two pieces of evidence can be aggregated using Dempster's rule of combination, assuming that the pieces of evidence are independent. The joint mass $m_{1,2}(\xi_t)$ assigned to each element ξ_t ($\xi_t \in X$) is:

$$m_{1,2}(\xi_t) = \frac{m_1(\xi_t)m_2(\xi_t) + m_1(\xi_t)m_2(\phi) + m_1(\phi)m_2(\xi_t)}{1 - K} \quad (4.1)$$

where $K = \sum_{i \neq j}^{i,j} m_1(\xi_i)m_2(\xi_j)$.

Note that $m_{1,2}(\xi_t)$ represents the combined belief for block A_t to be the root cause based on the two given pieces of evidence (two failing functional test stages).

Table 4.1: Combination of belief measures.

Belief functions \ Events	ξ_1	ξ_2	ξ_3	ϕ
\mathbf{m}_1	0.3	0.4	0.2	0.1
\mathbf{m}_2	0.5	0.3	–	0.2
$\mathbf{m}_{1,2}$	0.47	0.42	0.05	0.06

The expression $m_1(\xi_t)m_2(\xi_t) + m_1(\xi_t)m_2(\phi) + m_1(\phi)m_2(\xi_t)$ is the sum of all non-contradictory assignments to element ξ_t . It is further normalized by dividing with $1 - K$, where K is the total mass assigned to contradictory combinations.

Based on the evidence, we also have to update the reserved belief as follows:

$$m_{1,2}(\phi) = 1 - \sum_{\xi_t \in X} m_{1,2}(\xi_t) \quad (4.2)$$

Suppose we are given a board with three blocks A_1, A_2, A_3 , and a functional test for which a failure occurs. Suppose that we observe two failing stages. We know that each failing stage results in a belief function. Table 4.1 shows the combination of beliefs for each block to be the root cause. The events ξ_1, ξ_2, ξ_3 and ϕ are listed in the table. For example, ξ_1 represents the event that A_1 is the root cause and ϕ represents the event that no judgement is made. The first two rows show the belief assignments for each event based on the first and second failing stage, respectively. Note that “–” in the second row and the fourth column means that there is no assignment for ξ_3 in the belief function m_2 . The last row shows the aggregated belief assignment according to DS theory. The detailed calculation is as follows:

$$\begin{aligned}
K &= \sum_{\substack{i,j \\ i \neq j}} m_1(\xi_i)m_2(\xi_j) \\
&= m_1(\xi_1)m_2(\xi_2) + m_1(\xi_2)m_2(\xi_1) + m_1(\xi_3)m_2(\xi_1) \\
&\quad + m_1(\xi_3)m_2(\xi_2) \\
&= 0.3 \cdot 0.3 + 0.4 \cdot 0.5 + 0.2 \cdot 0.5 + 0.2 \cdot 0.3 = 0.45.
\end{aligned}$$

Next, we have

$$\begin{aligned}
m_{1,2}(\xi_1) &= \frac{m_1(\xi_1)m_2(\xi_1) + m_1(\xi_1)m_2(\phi) + m_1(\phi)m_2(\xi_1)}{1 - K} \\
&= \frac{0.3 \cdot 0.5 + 0.3 \cdot 0.2 + 0.5 \cdot 0.1}{1 - 0.45} = 0.47.
\end{aligned}$$

In the same way, we can calculate the combined (aggregated) belief for other events. From the last row of Table 4.1, we can see that ξ_1 has the highest combined belief. Therefore, we infer that block A_1 is the most likely suspect for this failure.

4.2 Ranking Candidate Blocks Based on Beliefs

The steps in the proposed method are as follows. First, the given functional test is partitioned into several stages. The partitioning is carried out based on functionality and the nature of the test. For each stage, we compare the test outcome for the board under test to golden (reference) values. In this way, we can determine the fail/pass status of each stage. Next, we use simulation to extract the dataflow graph for each stage by monitoring the toggling of specific signals. We assign belief to each block based on the dataflow graph and the pass/fail information. Finally, we use DS theory to calculate the combined belief for each block to be the root cause for each block,

```

module NetworkInterfaceTest();
    initial begin
        .....;
        Standard_configure;
        NetworkInterfaceInfoTest;
        AutoNegotiationTest;
        FrameTransferTest;
        TxPortStatsTest;
        pass_fail;
        $finish;
    end
endmodule

```

Figure 4.1: An example of a typical functional test sequence.

and the candidate blocks are ranked based on this information. The details of these steps are described as below.

4.2.1 Stage partitioning for the given functional test

A functional test sequence may contains millions of clock cycles. Figure 4.1 shows an example of a typical functional test sequence. It consists of several user-defined and system tasks. For example, in the Standard_configure task, the design under test is instantiated and clock/reset signals are made ready. After the system configuration is done, four test suites are run with various functionality. Each test is composed of sequences of commands, such as CPU read/write. After all the tests are applied, the pass_fail task is called to check the value of desired control and status registers (CSRs) and deliver the pass/fail message. Finally, the system task \$finish is invoked to terminate the test.

We can see that this functional test sequence can be easily divided into five stages. The first stage ends with the completion of the Standard_configure task. Next, each test task composes one stage. Suppose at the end of each stage, we can instantiate pass_fail task to check the values of related CSRs. Then we can easily determine whether this stage is a failing stage. The above method determines stages based on different test tasks. For a functional test that is composed of only one test task, we can still divide it into several stages by considering subsequences.

4.2.2 Dataflow extraction

The key concept here is the characteristic signal for a block under the given functional test. Consider a block, called *block_A*, and a functional test. Suppose that there are N signals, sig_1, sig_2, sig_N , in *block_A*. Suppose that when the given functional test is applied, these signals toggle x_1, x_2, x_N times, respectively. If the toggle count of sig_i , i.e., x_i , is greater than a threshold th when *block_A* is executed, sig_i is referred to as one of the *characteristic signals* for *block_A* under this functional test. The choice of th is determined by solving a classification-oriented optimization problem. The goal is to place the signals in two sets such that the sum of differences in the toggle counts between signals in these two sets is maximized.

Let $index(i, j)$ be defined as follows: $index(i, j) = 1$ if $x_i \leq th \leq x_j$ or $x_j \leq th \leq x_i$ for $i < j$; else $index(i, j) = 0$. Note that we can calculate th from the values of $index(i, j)$, $1 \leq i \leq N-1, i+1 \leq j \leq N$. The optimization problem can be described as follows.

$$\text{Maximize} \quad \sum_{i=1}^{N-1} \sum_{j=i+1}^N (index(i, j) \cdot |x_j - x_i|)$$

subject to

$$index(i, j) = \begin{cases} 1, & \text{if } (th - x_i)(th - x_j) \leq 0 \\ 0, & \text{otherwise} \end{cases}$$

The toggle count for a signal is known and it is obtained by running Verilog simulation in advance. By solving this optimization problem, we can determine th . Once th is determined, signals in this block can be categorized into two sets S_1 and S_2 . Each signal whose toggle count is above th , is placed in S_1 ; other signals are placed in S_2 . Signals in S_1 are the *characteristic signals* for this block. In this chapter, we use XPRESS-MP from FICO [94] to solve the optimization problem.

Table 4.2: Toggled characteristic signals in time slots of each stage.

Stages	Time slots	Toggled characteristic signals
stage1	<i>timeslot_1</i>	<i>sig1, sig2</i>
	<i>timeslot_2</i>	<i>sig31, sig4, sig5</i>
	<i>timeslot_3</i>	<i>sig6, sig1</i>
stage2	<i>timeslot_1</i>	<i>sig2</i>
	<i>timeslot_2</i>	<i>sig5</i>
	<i>timeslot_3</i>	<i>sig31, sig1</i>
stage3	<i>timeslot_1</i>	<i>sig3</i>
	<i>timeslot_2</i>	<i>sig6, sig4</i>

A time slot is defined as an interval of n consecutive clock cycles (n is a user-defined parameter). The parameter n can be chosen based on the length of the given functional test. In this chapter, we set n to be 100. We monitor the execution of blocks by observing their characteristic signals in each time slot. Each block can be associated with several characteristic signals.

Suppose a functional test is partitioned into three stages, namely *stage1*, *stage2*, and *stage3*. Suppose *stage1* and *stage2* include three time slots, and *stage3* includes two time slots. We monitor the characteristic signals and check whether their values toggle in each time slot in each stage. Table 4.2 lists the toggled characteristic signals in each time slot for every stage.

Typically, there is dataflow between blocks when a functional test is applied to a board, i.e., data is propagated from the outputs of one block to the inputs of another block. We consider the dataflow from the blocks that are active in one time slot to blocks that are active in the subsequent time slot. Based on the accumulated information for the active blocks executed in each time slot for a stage, we can create the dataflow graph for this stage (We ignore the dataflow between blocks that are active in the same time slot.). For example, Figure 4.2(a) shows the dataflow graph for *stage1* of the given functional test, generated based on Table 4.2. A total of three time slots are included in this stage. Blocks that are active in each time slot are

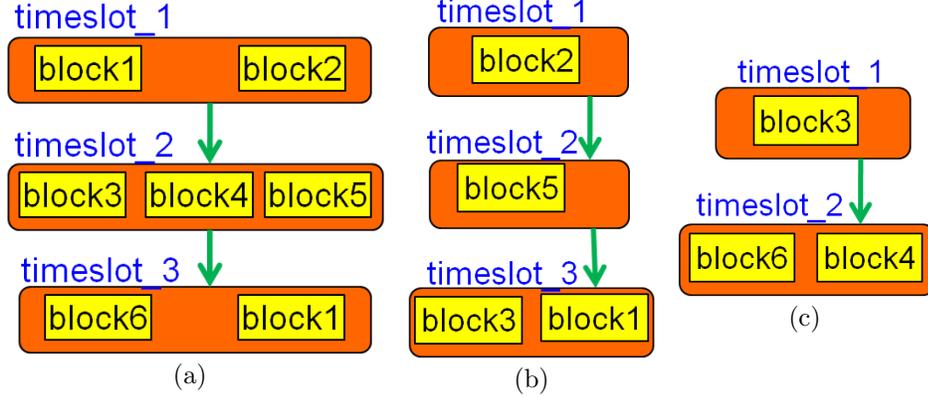


Figure 4.2: Dataflow graph for: (a) *stage1*; (b) *stage2*; (c) *stage3*.

shown within the rectangle corresponding to this time slot. An arrow in the dataflow graph indicates the flow of data. Similarly, the dataflow graphs for *stage2* and *stage3* are shown in Figure 4.2(b)-(c).

4.2.3 Assignment of belief based on each failing stage

It is difficult to infer the root cause when no error is observed. Therefore, passing stages are less important than failing stages (i.e., provide less information) in our approach than failing stages for the ranking of candidate blocks. A passing stage may only imply that the fault is not activated or observed, while a failing stage can provide strong evidence for corresponding blocks to be the root cause. In this work, we calculate belief measures and rank blocks based only on information about failing stages.

Suppose a board is composed of N blocks A_1, A_2, \dots, A_N . We associate a weight to each block in order to assign belief more accurately. The weight can be obtained on the basis of metrics such as the attributes of the design, the given functional test, and the blocks. In this work, we use the number of register bits in each block to represent its size. Our premise is that a larger block is more likely to be the root cause of a failure, thus it is assigned a larger weight and a larger belief measure. RTL models

and information about registers in each block are available from board designers. Functional tests typically come from design verification suites. During board-level verification, board designers have exact RT-level descriptions for some components (such as the components designed internally) and at least sufficient information (interfaces and behavioral description) related to other components for running verification test sequences. Therefore, it is possible to simulate functional tests at a board-level environment, and target the test and diagnosis of those components with exact RT-level descriptions. For example, this information was provided to us for the test case when we were generating experimental data. If the above details are not available, equal weights can be assigned to the blocks. Suppose the number of register bits for the N blocks are S_1, S_2, \dots, S_N , respectively. Let $S_{max} = \max\{S_1, \dots, S_N\}$. We define the weight of block A_i to be S_i/S_{max} . In this way, we obtain the weights for N blocks, denoted by w_1, w_2, \dots, w_N . We not only consider the size of blocks, but also monitor the dataflow. If there is no dataflow through the largest blocks, they will still be ranked very low. Therefore, the largest blocks are ranked not only on the basis of their sizes, but also on the basis of dataflow.

Each failing stage and dataflow graph results in a belief function m . The belief assignment obeys the following rules:

1. Some belief is reserved for the event ϕ (no conclusion is made), i.e., $m(\phi) = 1 - M/N$, where M is the number of blocks executed in this stage;
2. Each time slot in this stage owns an equal portion of the remaining belief, i.e., $(1 - m(\phi))/n_{ts}$, where n_{ts} is the number of time slots in this stage;
3. Blocks executed in the same time slot share the belief of this time slot according to their weights. For example, suppose in the i th time slot, t blocks $(A_{i1}, A_{i2}, \dots, A_{it})$ are executed. Then in this time slot, the belief for each executed

block is

$$m_i(A_{ij}) = \frac{1 - m(\phi)}{n_{ts}} \times \frac{w_{ij}}{w_{i1} + w_{i2} + \dots w_{it}}, j = 1, \dots, t;$$

4. The final belief assigned to each block based on this failing stage is: $m(A_k) = \sum_{i=1}^{n_{ts}} m_i(A_k), k = 1, 2, \dots, N.$

Let us take Figure 4.3 as an example. Here *stage1* is a failing stage and it includes 3 time slots. Since all 6 blocks have been executed in this failing stage, we assign $m(\phi) = 0$. Therefore, each time slot owns a belief of $1/3$. Suppose the weights for *block1*, *block2*..., *block6* are $w_1, w_2, w_3, w_4, w_5, w_6$, respectively, where $\{w_1, w_2, w_3, w_4, w_5, w_6\} = \{0.5, 1, 0.2, 0.3, 0.4, 0.7\}$. In *timeslot_1*, *block1* and *block2* are executed. Thus, the belief of $1/3$ is divided between them according to their weights. The belief for *block1* in this time slot is $1/3 \times 0.5/(0.5 + 1)$, i.e, $1/9$. The belief for *block2* in this time slot is $1/3 \times 1/(0.5 + 1)$, i.e, $2/9$. Similarly, we can compute the belief for each block in each time slot, as shown in Figure 4.3. The final belief assigned to each block in this stage is the aggregate of its belief in all time slots. Therefore, in this example, we have $m(\text{block1}) = 1/9 + 5/36$, i.e., $1/4$; $m(\text{block2}) = 2/9$; $m(\text{block3}) = 2/27$; $m(\text{block4}) = 1/9$; $m(\text{block5}) = 4/27$; $m(\text{block6}) = 7/36$.

In this way, we can compute the belief that a block is the root cause based on each failing stage.

4.2.4 Combination of beliefs and ranking of blocks

After assigning belief measures based on each failing stage, we combine them using the Equations (4.1)-(4.2) and obtain the final combined belief for each block to be

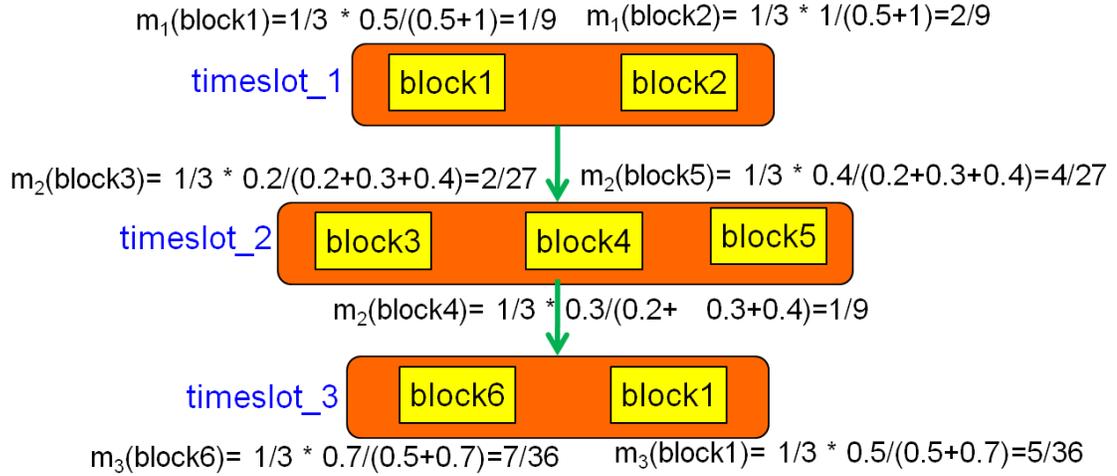


Figure 4.3: Belief assignment based on *stage1* of the given functional test.

the root cause for each block, on the basis of which the candidate blocks are ranked. The blocks with higher beliefs are more likely to be the root cause for the board-level functional failure and therefore are ranked at the top. By targeting blocks in this ranked order, we can significantly reduce the time needed for more refined diagnosis.

4.3 Experimental Evaluation

The proposed method is applicable to components on a board as well as modules within components. To evaluate its effectiveness, we performed experiments on an industry design (called *design_A* in this chapter). This design is a multi-port Ethernet network interface, and it has been implemented using 65 nm technology. In the final gate-level netlist, there are about 176K flip-flops and 4M gates. A test board has been built by putting two instances of *design_A* together. The functional test comes from the design-verification suite, which is provided by its developers. Our goal is to show that the proposed method can provide an effective ranking of the candidate blocks when a board-level functional failure occurs.

4.3.1 Overview of *design_A*

The design includes the NETWORKINTERFACE unit, forwarding controller logic, buffering logic/memory, and queuing logic/memory necessary to provide switching and classification among different network layers. In our experiments, the given functional test targets the functionality of the NETWORKINTERFACE unit. There are a total of 32 blocks in this unit. We inject faults in these blocks to create artificial faulty boards in our experiments and our goal is to rank these candidate faulty blocks. To ensure that board-level failures are considered, the functional test is applied in a board-level environment instead of at the inputs of the NetworkInterface unit. We limit the root cause of this board-level functional failure to reside in the blocks of the NetworkInterface unit since the given functional test targets only this unit.

4.3.2 Simulation results

An artificial faulty board is created by inserting a stuck-at fault, a bit-flip fault, or a delay fault at board level within one of the blocks. Even though high stuck-at coverage is typically obtained for chip testing, we consider stuck-at faults as a simple approach to model additional board-level failures. Bit flips model transient faults, and we consider delay faults on non-critical paths that are not targeted for chip testing but might cause fails at board level due to power-supply noise, clock skew, and crosstalk on the board. We collect the pass/fail information for each stage of the given functional test, obtain the dataflow graph for each stage, calculate the belief measure for a block to be the root cause, and finally rank the blocks. We then verify whether our ranking is accurate for the injected faults.

The given functional test is partitioned into 11 stages according to its functionality (Table 4.3). We monitor the characteristic signals of blocks in each time slot, based on which the dataflow graph is created for each stage. All experiments were performed

Table 4.3: Stages of the given functional test for *design_A*.

Stage number	Stage name	No. of time slots
1	basic_test	9
2	auto_negotiation_test	9
3	high_speed_mode_transfer_test	194
4	low_speed_mode_transfer_test	266
5	high_speed_mode_tx_stats_test	69
6	high_speed_mode_rx_stats_test	31
7	half_duplex_test	44
8	low_speed_mode_tx_stats_test	607
9	mixing_traffic_test	13
10	rx_fifo_test	380
11	rx_short_test	10

on a pool of 64-bit Linux servers. Synopsys Verilog Compiler (VCS) was used to run Verilog simulation. All other programs were implemented in C++ and Perl scripts.

Results for stuck-at faults

We create 100 faulty boards by injecting stuck-at faults in the blocks. One fault is injected for each faulty board. In a faulty board, the block with the resident fault is called the target block. We consider 10 different blocks as the target blocks.

We collect the pass/fail information of all stages for one faulty board due to a stuck-at fault in approximately 1.9 hours of CPU time. A total simulation time of 190 hours is required for all the 100 faulty boards. It takes only a few seconds to assign belief based on failing stages and calculate the combined belief for each block to be the root cause. Therefore, while the simulation time for validating the method is high, the CPU time for actual ranking of the blocks based on pass/fail information is negligible. Therefore, the proposed method is practical for large designs.

For each faulty board, DS theory is used to obtain the combined belief that each of the 32 candidate blocks is the root cause. In the ideal case, the target block for this faulty board is expected to have the highest belief and be ranked first. However, sometimes blocks other than the target block will have the highest belief, which

Table 4.4: Results on ranking for failures due to the injected faults.

Name of target blocks	No. of faulty boards	No. of boards for which ranking is effective				
		stuck-at			bit-flip	delay
		top-3 ($k = 3$)	top-2 ($k = 2$)	top-1 ($k = 1$)	top-3 ($k = 3$)	top-3 ($k = 3$)
Global_Regs	10	8	8	6	9	7
Addr_Decoder	10	9	4	0	9	9
Rx_Scheduler	10	8	0	0	7	6
Tx_Scheduler	10	8	0	0	7	7
Status_Scheduler	10	6	2	0	2	5
Data_Mux	10	10	10	10	10	10
Hi_Speed_Port0	10	10	9	3	10	9
Hi_Speed_Port1	10	9	8	0	9	8
Lo_Speed_Port2	10	10	7	0	10	9
Lo_Speed_Port3	10	9	9	0	10	8

necessitates the inclusion of additional highly-ranked blocks in the list of suspects. We describe the ranking to be effective if the target block belongs to one of the top- k blocks (k is a small user-defined number).

Table 4.4 shows the ranking results for failures due to the injected faults. For example, Row 6 and Column 3 show that, for the 10 faulty boards with *Global_Regs* being the target block, we obtain effective ranking in 8 cases if we consider the top-3 blocks. From Table 4.4, we can see that if we only consider $k = 1$ and $k = 2$, the number of effective rankings is small in some cases. However, if we consider $k = 3$, the proposed method provides highly effective rankings for most faulty boards. The time needed for diagnosis in the second phase is significantly reduced since we can identify the root cause by targeting only 3 blocks instead of all 32 blocks. The choice of k depends on the number of candidate blocks and a characterization of this parameter is left for future work.

Results for bit-flip faults

Transient faults are a serious concern at the board level [95]. Since the effects of transient faults are subtle and failures due to transient faults are not always reproducible, simulation for each bit-flip fault is run multiple times and an average is used to compute the belief measure. In our experiments, simulation is performed 30 times for each bit-flip fault. During these 30 runs, the bit-flip fault is inserted at randomly chosen time instants during test execution. For each bit-flip fault, the failing ratio of a stage is defined as number of times that this stage fails divided by the total number of simulations. If the failing ratio of a stage is more than 50% for a bit-flip fault, we record this stage as a failing stage for this fault. Next, a belief is assigned based on this failing stage as in the case of stuck-at faults and can be further combined with beliefs resulted from other failing stages.

We collect the pass/fail information of all stages for one faulty board due to a bit-flip fault in approximately 57 hours of CPU time. We also create 100 faulty boards by injecting bit-flip faults. It takes only a few seconds to assign belief based on failing stages and calculate the combined belief for each block to be the root cause. We can see similar results as in the case of stuck-at faults, i.e, the proposed method provides effective rankings for most faulty boards.

Results for delay faults

Power supply noise is considered as one of the major causes for board-level functional failures [86]. The ATE test environment and board-level functional test environment tend to differ in terms of power supply and clock characteristics. In ATE test, only a few clock cycles (typically one or two clock cycles) are considered for response capture. On the other hand, board-level functional test involves millions of clock cycles, which is close to operation in functional mode. Therefore, a board-level functional test can

often detect delay faults due to power-supply noise and crosstalk that are missed by the ATE during chip testing.

We simulate board-level functional failures due to delay faults that escape chip-level testing. Since delays on critical paths or paths with small slack are targeted in chip testing, we only consider delay faults on shorter paths that might become critical on the board due to noise. We target the scenario that board-level noise results in additional delays on non-critical paths if a large fraction of signals in the design have high switching activities (we assume that at least 90% of the signals toggle in one clock cycle) within a given window of clock cycles (10 consecutive cycles in our work).

We create a faulty board by injecting a delay on 10 short paths within a block (identified by static timing analysis) during test execution when the switching activities of signals meet the condition described above. We collect the pass/fail information of all stages for each faulty board in approximately 2.5 hours of CPU time. A belief is assigned based on this failing stage as in the case of stuck-at faults and can be further combined with beliefs resulted from other failing stages. It takes only a few seconds to assign belief based on failing stages and calculate the combined belief for each block to be the root cause. We can see similar results as in the case of stuck-at faults, i.e, the proposed method provides effective rankings for most faulty boards. This highlights the effectiveness of the proposed ranking method based on dataflow-analysis and DS theory.

We compared the proposed method to a reasoning technique based on Bayesian inference, which requires fault simulation to generate a list of syndromes [92]. The accuracy of candidate ranking was similar for the two methods (the method based on DS theory performed slightly better for five target blocks, while Bayesian inference was slightly better in the other five cases), but Bayesian inference required an excessive amount of CPU time for generating the data needed for analysis. The advantage

of the DS-based method is that when an error is observed for a board, we only need to run logic simulation to obtain the pass/fail information of the functional-test stages for analysis. In contrast, for Bayesian inference, before we can perform analysis, we need to obtain a set of conditional probabilities through fault simulation runs. In our experiments, the CPU time needed to obtain all this data for Bayesian inference was 100 times that for the proposed method. Therefore, the proposed approach is promising for large and complex boards with many components and a large number of modules within components.

4.4 Chapter Summary and Conclusions

In this chapter, we have proposed a new method to rank candidate faulty blocks based on dataflow analysis and DS theory for the diagnosis of board-level functional failures. The proposed method transforms the information of a functional test failure into the information of multiple-stage failures by partitioning the given functional test into multiple stages. A belief measure is assigned to each block based on each failing stage and DS theory is used to combine the beliefs from multiple failing stages. Experiments for an industry design show that the proposed method provides effective rankings for most of the board-level functional failures due to injected faults.

Chapter 5

Design-for-Testability and Diagnosis Methods for Board-Level Functional Failures

In this chapter, we focus on finding the root cause of NTF. We propose an innovative functional test approach and DFT methods for the detection of board-level functional failures. These DFT and test methods allow us to reproduce and detect functional failures in a controlled deterministic environment, which can provide ATE tests to the supplier for early screening of defective parts. Experiments on an industry design show that the proposed functional scan test with appropriate functional constraints can adequately mimic the functional state space, as measured by appropriate coverage metrics. Experiments also show that most functional failures due to stuck-at, dominant bridging, crosstalk and delay faults due to power supply noise can be reproduced and detected by functional scan test. We also describe two approaches to enhance the mimicking of the functional state space. The first approach allows us to select a given number of initial states in linear time and functional scan tests resulting from these selected states are used to mimic the functional state space. The second approach is based on controlled state injection.

5.1 Motivation and Problem Statement

We use Figure 5.1 as a motivating example. The dots represent the states close to a functional state. Using a functional state or a close-to-functional state as initial state, we can run controlled, deterministic test for a reasonable number of clock cycles and obtain a new deterministic-test state space S . Intuitively, if we can use this state

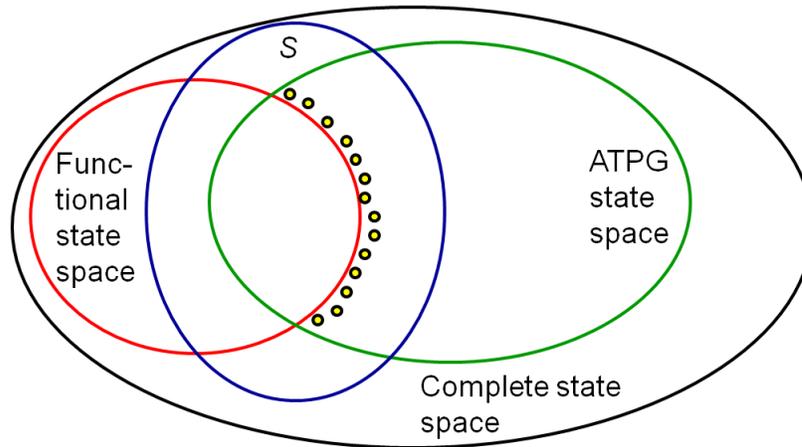


Figure 5.1: An illustration of the state space for different test methods.

space to mimic the functional state space, it is possible for us to use deterministic test to reproduce and detect functional failures.

For example, suppose a fault occurs when a circuit traverses state S_1 in functional test mode. It is possible that this failure will also be detected by deterministic test if S_1 belongs to the deterministic-test state space. Furthermore, let S_1 be the state 1101 and \bar{S}_1 be another state 1100. Since the Hamming distance between these two states is small, there is high likelihood that the fault will be detected in deterministic test if \bar{S}_1 belongs to the deterministic-test state space.

In this chapter, we focus on solving two problems. The first problem is stated as follows:

Given: A board design including several components;

Goal: Add DFT hardware to each component of the board, such that we can

- provide controlled and deterministic internal clocks;
- lock the circuit state of a component when a functional test fails. The locking can include multiple states before the test fails. The states are determined by the flip-flops in each component;
- dump the desired circuit state;

- recover to a desired circuit state.

The second problem can be stated as follows.

Given: We are given the following:

- a board design with each component having the characteristics described in the first problem;
- the components of the board pass all structural tests;
- the board fails when a functional test is applied;
- there is a golden board available that passes the target functional test;
- the root cause of the functional failure has been located to a specific faulty component.

Goal: Reproduce and detect the functional failure in a controlled deterministic environment and obtain the final test that can be returned to the supplier side and ported to ATE.

5.2 Functional Scan Design and Functional Scan Test

5.2.1 Functional scan design

To address the first problem, we add functional scan design to each component of the target board. The hardware implementation of functional scan design includes four parts: clock controller, circuit state load and unload, memory contents load and unload, and pseudorandom bit sequence (PRBS) generator. Fig. 5.2 shows an example of functional scan design for a ASIC, namely ASIC1.

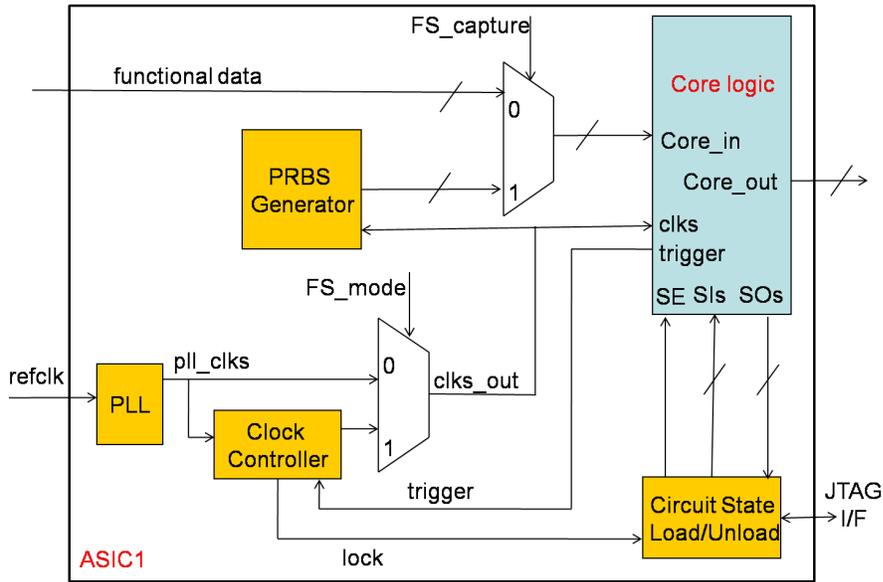


Figure 5.2: An example of functional scan design.

Clock controller

The clock controller is a key module that is needed to support the locking and recovering of a component's circuit state when board-level functional failure occurs. It allows us to start counting and stop the clock at predetermined times. The On/Off of clock pulses is controlled by user-defined trigger signals (such as hardware interrupt or ECC failure) and a programmable counter. For multiple clock domain designs, the clock controller provides the flexibility to adjust clock phase and the latency relationship to mimic the relationship between clocks in functional test. Consider Fig. 5.2 as an example. Given the reference clock *refclk*, the PLL unit generates free running PLL clocks *pll_clks*. The signal *FS_mode* indicates whether the system is in functional scan mode. The signal *trigger* is used to indicate the beginning of propagating clock pulses of *pll_clks* to the output clocks *clks_out*. In functional scan mode, once *trigger* is asserted, the counter inside the clock controller begins to count. After it has counted for a desired number of clock cycles, the clock controller stops propagating clock pulses of *pll_clks* to *clks_out* and the output *lock* is asserted. (Note

that *clks_out* may consist millions of clock cycles thus the power/noise condition of functional scan mode is quite close to that of functional mode.) Since all the clocks to the Core Logic unit will be stopped and there will be no pulses on them when *lock* is asserted, we can easily lock the circuit state for dumping.

Circuit state load and unload

This hardware provides the capability to load and unload the circuit state using scan chains. When a board fails in functional test, we use the unload feature to dump the locked functional circuit state of the faulty component. Next we use the load feature to recover the functional circuit state of the component in a controlled deterministic environment. Moreover, the unload feature can also be used to lock and dump the circuit states in the controlled deterministic environment. In Fig. 5.2, there is a JTAG interface for the circuit state load and unload unit. Under the control of the signal to indicate a beginning of load procedure, which comes from the JTAG interface, and the control of the signal *lock*, the circuit state load/unload unit generates the scan enable signal *SE* and feeds it to the Core Logic. When the signal *SE* is asserted, the Core Logic begins to load or unload the circuit state through the scan chain ports *SIs* or *SOs*.

PRBS generator

A PRBS is a statistically random bit sequence. It is usually generated using a linear-feedback-shift-register (LFSR). A PRBS is useful for mimicking the random nature of defects due to the package, data I/O, and the power/noise variation. Here we use a PRBS generator to generate the pseudorandom patterns for the inputs of the faulty component for use in diagnosis. In Fig. 5.2, the outputs of PRBS generator and the normal functional data are selected to feed the primary inputs (PIs) of the

Core Logic unit through a MUX, which is controlled by *fs_capture*. When it is 1, the values for PRBS generator are passed to *Core_in* of the Core Logic.

5.2.2 Functional scan test

Based on the functional scan design, we now define a new type of test, namely functional scan test. It is defined to be a test that involves the following procedure:

- 1) The board enters functional scan mode.
- 2) We bring the target component to an initial state that is close to a functional state using the load capability. (The “closeness” metric is quantified in Section 5.3.) Ideally, the initial state is a functional state. However, it is hard to lock and recover to an exact functional state due to the existence of memories.
- 3) Vary PIs and internal flip-flops values of the target component and run desired capture cycles, where the capture clock is from the customized clock controller. In this chapter, we only vary values of PIs.
- 4) Dump and observe the circuit state and primary outputs (POs) of the target component using the unload capability.

Figure 5.3 shows an example of clock timing for a functional scan test. The signals *pll_clk1* and *pll_clk2* are free running clocks from the PLL. The signal *TCK* is the slow clock from JTAG. The clocks *clk1* and *clk2* are fed to internal cells. In functional scan program mode, we configure the appropriate parameters for the faulty component of the board, such as the number of capture clock cycles, which clock domains to enable, and so on. Next, the *FS_mode* signal is asserted and the board enters functional scan load mode. In this mode, the target component is brought to the desired initial state. Next, the *FS_capture* signal is asserted and the

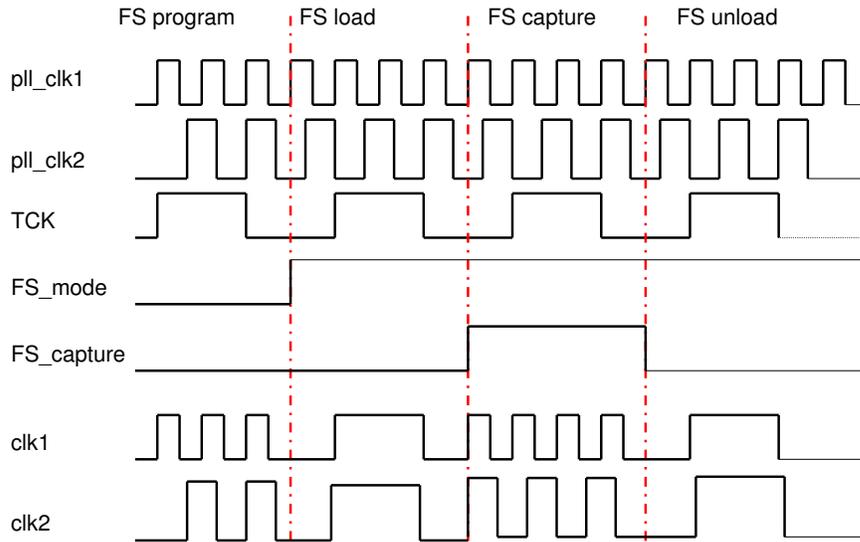


Figure 5.3: An example of clock timing for functional scan test.

board enters functional scan capture mode. In this mode, we vary PIs and internal flip-flops values and run a desired number of capture cycles. Finally, the *FS_capture* signal is de-asserted and the board enters functional scan unload mode. In this mode, the values of the circuit state and POs are dumped and observed. In the functional scan program mode and functional scan capture mode, clocks to internal cells *clk1* and *clk2* come from the PLL clocks. In the functional scan load and unload mode, *clk1* and *clk2* come from the slow clock *TCK*.

Note that the capture procedure of functional scan mode may consist of millions of cycles and the capture clocks are similar to that of normal functional mode. In view of this, functional scan test has a similar clocking scheme compared to functional test. Thus, functional scan test provides similar test conditions as functional test, involving power supply, voltage levels, temperature, and timing. Also, functional scan test can cover the exactly same paths as that covered by functional test, some of which may not be covered in traditional ATPG mode. We can therefore see that functional scan test bridges the gap between functional test and ATPG test. It provides the controllable and deterministic characteristics of ATPG test. On the

other hand, it uses the same capture path as in functional test. It can thus provide a test condition close to functional test in a controlled manner. Once the functional failure has been reproduced and detected in functional scan test, it will be easier to perform the subsequent diagnosis procedure as well as recover and port the test to ATE due to the controllable and deterministic characteristics of functional scan test.

5.3 Key Definitions

In this section, we introduce some definitions that are used to evaluate the effectiveness of functional scan test.

Degree of similarity between two circuit states: Given two states S_i and S_j (with the same number of bits in the binary encoding), the degree of similarity between S_i and S_j , $SD(S_i, S_j)$, is defined as $M(S_i, S_j)/B(S_i)$, where $B(S_i)$ is the number of bits in S_i and $M(S_i, S_j)$ is the number of matched bits between S_i and S_j .

For example, suppose S_1 is 1101 and S_2 is 1100. Then $B(S_1)$ is 4 and $M(S_1, S_2)$ is 3 (the first three bits are matched). Thus the similarity degree $SD(S_1, S_2)$ is 3/4.

State sequence of depth k : Suppose that in k consecutive clock cycles, the circuit state changes as follows: $S_1 \rightarrow S_2 \rightarrow S_3 \dots \rightarrow S_k$. Then $S_1 \rightarrow S_2 \rightarrow S_3 \dots \rightarrow S_k$ is defined to be a state sequence of depth k .

Degree of similarity between two state transitions: Given two state transitions $S_i \rightarrow S_j$ and $\bar{S}_i \rightarrow \bar{S}_j$, the degree of similarity between them is defined as:

$$SD(S_i \rightarrow S_j; \bar{S}_i \rightarrow \bar{S}_j) = \frac{SD(S_i, \bar{S}_i) + SD(S_j, \bar{S}_j)}{2}. \quad (5.1)$$

Degree of similarity between two state sequences: Given two state sequences of depth k : $S_1 \rightarrow S_2 \dots \rightarrow S_k$ and $\bar{S}_1 \rightarrow \bar{S}_2 \dots \rightarrow \bar{S}_k$, their degree of similarity is

defined as

$$DSS(S_1 \dots \rightarrow S_k; \bar{S}_1 \dots \rightarrow \bar{S}_k) = \frac{\sum_{i=1}^{i=k} SD(S_i, \bar{S}_i)}{k}. \quad (5.2)$$

For a given functional test, we can obtain the set of functional states, the set of functional state transitions, and the set of functional state sequences of depth k for any given k . In order to check whether the functional scan state space can adequately mimic the functional state space, we define the following three terms: α _coverage of functional states, α _coverage of functional state transitions, and α _coverage of functional state sequences.

α _coverage of functional states: Suppose we are given a threshold α ($0 \leq \alpha \leq 1$), where α is a user-defined parameter. For a functional state S_i in the functional state set, if we can find a state \bar{S}_i in the set of functional scan states such that $SD(S_i, \bar{S}_i) \geq \alpha$, we say that the functional state S_i is covered with degree of similarity α . Suppose the number of functional states covered with degree of similarity α is $n(state_alpha)$ and the number of states in the functional state set is $n(func_state)$. The α _coverage of functional states is defined by:

$$Cov_S_alpha = \frac{n(state_alpha)}{n(func_state)} \quad (5.3)$$

α _coverage of functional state transitions: Once again, suppose we are given a threshold α ($0 \leq \alpha \leq 1$), where α is a user-defined parameter. Consider a functional state transition $S_i \rightarrow S_j$ in the set of functional state transitions. If we can find another state transition $\bar{S}_i \rightarrow \bar{S}_j$ in the set of functional scan state transition such that $SD(S_i \rightarrow S_j, \bar{S}_i \rightarrow \bar{S}_j) \geq \alpha$, we say that the functional state transition $S_i \rightarrow S_j$ is covered with degree of similarity α . Suppose the number of functional state transition covered with degree of similarity α is $n(statetrans_alpha)$ and the number of state transitions in the set of functional state transition is $n(func_state_trans)$. The

α _coverage of functional state transition is defined as follows:

$$Cov_ST_alpha = \frac{n(state_trans_alpha)}{n(func_state_trans)} \quad (5.4)$$

α _coverage of functional state sequence of depth k : Consider a functional state sequence $S_1 \rightarrow S_2 \dots \rightarrow S_k$ of depth k . If we can find a state sequence $\bar{S}_1 \rightarrow \bar{S}_2 \dots \rightarrow \bar{S}_k$ of depth k in the set of functional scan state sequence such that $DSS(S_1 \rightarrow S_2 \dots \rightarrow S_k, \bar{S}_1 \rightarrow \bar{S}_2 \dots \rightarrow \bar{S}_k) \geq \alpha$, we say that the functional state sequence $S_1 \rightarrow S_2 \dots \rightarrow S_k$ is covered with degree of similarity α . Suppose the number of functional state sequence of depth k covered with degree of similarity α is $n(state_seq_k_alpha)$ and the number of state sequences of depth k in the set of functional state sequence is $n(func_state_seq_k)$. The α _coverage of functional state sequence of depth k is defined as follows:

$$Cov_SS_k_alpha = \frac{n(state_seq_k_alpha)}{n(func_state_seq_k)} \quad (5.5)$$

Our premise is that if we can obtain high Cov_S_alpha , high Cov_ST_alpha , and high $Cov_SS_k_alpha$ for a large value of α , the functional scan state space can mimic the functional state space well. This implies that the functional scan test adequately reproduces the functional failures.

5.4 Reproduction and Detection of Board-Level Functional Failures

In this section, we describe how functional scan test is used to reproduce and detect functional failures. Three major steps are included in the proposed flow:

Step 1: Divide the functional test into stages and find the first failing stage. A functional test sequence may contains millions of clock cycles. We first partition the

given functional test sequence into several stages according to its functionality. At the end of each stage, we check the values of related control and status registers (CSRs). In this way, we can easily identify the first failing stage for the given functional test.

Step 2: Obtain the initial cycle and dump initial state in functional test. We first define the following terms.

Faulty cycle: Suppose a failing stage contains N cycles C_1, \dots, C_N . Suppose at the strobe point of C_i ($i = 1, 2, \dots, N$), the good board and faulty board have different values on observation points (CSRs or scan flip-flops or POs). Then we refer to C_i as a faulty cycle.

k -cycle fault: If k cycles are needed to activate a fault and capture the fault effect, the fault is referred to as a k -cycle fault.

Initial cycle and initial state: Suppose C_i is a faulty cycle in a functional test. The initial cycle of C_i is C_j , where j equals $i - k$ for a k -cycle fault. The circuit state corresponding to the initial cycle is defined as the initial state.

It is hard to apply functional vectors on the ATE due to the following reasons: 1) High cost of functional vector translation/transformation; 2) Strict timing constraints on ASIC boundary signals are required, which requires a high-capacity tester. Therefore, we propose to use functional scan test to reproduce the failure. In order to do so, we first use binary search to find a narrowed-down faulty cycle in functional test for the first failing stage. Then we find the initial cycle corresponding to this faulty cycle. Next, we dump the circuit state of the initial cycle from the good board. This dumped state will be used in the next step to reproduce the functional failures.

Step 3: Reproduce and detect functional failures in functional scan test. We next reproduce and detect the functional failure in functional scan test starting from the initial state (dumped state in Step 2). By running functional scan test with varying PI values and applying appropriate functional constraints (extracted according to

the given functional test and the specification of the design) for a desired number of clock cycles, we can check whether the functional failure is reproduced in functional scan test.

5.5 Experimental Evaluation

To evaluate the proposed method, we performed experiments on an industry design (called *design_A* in this chapter). The industry design is a multi-port Ethernet network interface to the HULC stacking system. It has been implemented in 65 nm technology. In the final gate-level netlist, there are about 176K flip-flops and 4M gates. A test board has been built by putting two instances of *design_A* together. The functional test for *design_A* comes from the design-verification suites, provided by its developers.

Our first goal is to show that with proper functional constraints, functional scan test can adequately mimic the functional state space, measured by Cov_{S_α} , Cov_{ST_α} , and $Cov_{SS_k_\alpha}$. Recall that this is the theoretical basis for using functional scan test to reproduce and detect functional failures. Our second goal is to show that the proposed flow can efficiently reproduce and detect functional failures by using functional scan test. Thus it can be used to return the narrowed-down test to the supplier for ATE use.

5.5.1 Experimental setup

All experiments were performed on a 64-bit Linux server with 4 GB memory. Synopsys Verilog Compiler (VCS) was used to run Verilog simulation. All other programs were implemented using C++ and Perl scripts.

5.5.2 Mimicking of functional state space

Substate group partition

It is impractical to consider the complete state space of a board or even the complete state space of a component due to design complexity. (The total state space is of size 2^n for a design with n flip-flops.) Furthermore, sometimes it is unnecessary and meaningless to consider all the states. For example, suppose the state space of a design is determined by four flip-flops: $ff1, ff2, ff3, ff4$. Suppose $ff1, ff2$ are related to one function (such as a read operation) of the design and $ff3, ff4$ are related to another function (such as a counter). In this case, instead of considering the complete state space, it is more meaningful to partition the four flip-flops into two substates ($ff1, ff2$ in group 1 and $ff3, ff4$ in group 2) and measure the state coverage of the two groups separately.

In [99] [100], substate partitioning methods based on circuit structure have been proposed to guide ATPG. However, this partitioning method relies on the circuit structure and it does not consider in functional mode; therefore it only benefits structural ATPG test.

In this chapter, we determine the substate groups at register transfer (RT)-level by considering the functionality of the design. Our key idea is to place registers that share common conditions for assignment operations in one substate group. First, the assignment condition for each register is extracted from the RT-level description. We extract three types of assignment conditions:

- 1) Posedge or negedge of clocks. For example, in Figure 5.4, register *data* is assigned value at the posedge of *sysClk*. Therefore, we can extract the assignment condition for *data* as: $\{sysClk @posedge\}$.
- 2) Signals in sensitive list. For example, from Figure 5.4, we can extract the assign-

```

always @ (posedge sysClk) begin
    data[7:0] <= data_keep[7:0];
end

always @ (valid or ready ) begin
    nextStage = Stage;
if (valid) begin load=1'b1; end
end

```

Figure 5.4: Examples for extracting assignment conditions.

```

always @ (posedge sysClk) begin
    data[7:0] <= data_keep[7:0];
end

always @ (valid or ready ) begin
    nextStage = Stage;
if (valid) begin
    load=1'b1;
    ack=1'b1;
end
end

always @ (ready or select ) begin
    load=1'b0;
    port1_en=1'b1;
end

```

Figure 5.5: An example of substate group partitioning.

ment condition for register *nextStage* as: $\{valid, ready\}$.

- 3) Specific signal value. Again, from Figure 5.4, we can see that the register *load* is assigned value when *valid* is 1, besides the sensitive signals list. Thus we can extract the assignment condition for register *load* as: $\{valid, ready, valid = 1\}$.

Let us use an example to illustrate substate group partitioning. Consider the RTL description shown in Figure 5.5. The assignment condition for each register can be extracted as shown in Table 5.1. Based on Table 5.1, we can partition the state space into three groups: $group1 = \{data\}$, $group2 = \{nextStage, load, ack\}$, $group3$

Table 5.1: Extracted assignment conditions for registers from Fig. 5.5.

Register name	Assignment conditions
<i>data</i>	{ <i>sysClk</i> @posedge}
<i>nextStage</i>	{ <i>valid</i> , <i>ready</i> }
<i>load</i>	{ <i>valid</i> , <i>ready</i> , <i>valid</i> = 1}
<i>ack</i>	{ <i>valid</i> , <i>ready</i> , <i>valid</i> = 1}
<i>load</i>	{ <i>ready</i> , <i>select</i> }
<i>port1_en</i>	{ <i>ready</i> , <i>select</i> }

= {*load*, *port1_en*}. The register *nextStage* is placed in group2 because it shares the common condition {*valid*, *ready*} with *load* and *ack*, even though the condition is not exactly the same. The register *port1_en* is not placed in group2 because its assignment condition is {*ready*, *select*}, but *select* is not included in the condition for group2. Note that register *load* belongs to both group2 and group3.

Simulation results

First, we partition the state space into several substate groups. Next, for each substate group, we record the number of unique functional states, state transitions, and state sequences traversed by the given functional test. Then, we apply functional scan test and calculate the α -coverage for functional states, state transitions, and state sequences. In functional scan test, we use a PRBS generator to generate patterns for inputs of *design_A*. Also, for *design_A*, we extract and apply functional constraints during functional scan test. Finally, we observe whether we can obtain high values of α -coverage metrics with a large value of α .

For *design_A*, we consider 63 substate groups that are related to the given function test. The number of flip-flop bits for 53% of the substate groups belongs to the range of 10-49. The functional test for *design_A* includes 29,317 clock cycles and the functional scan test has 100,000 capture cycles. In functional scan test, we use a

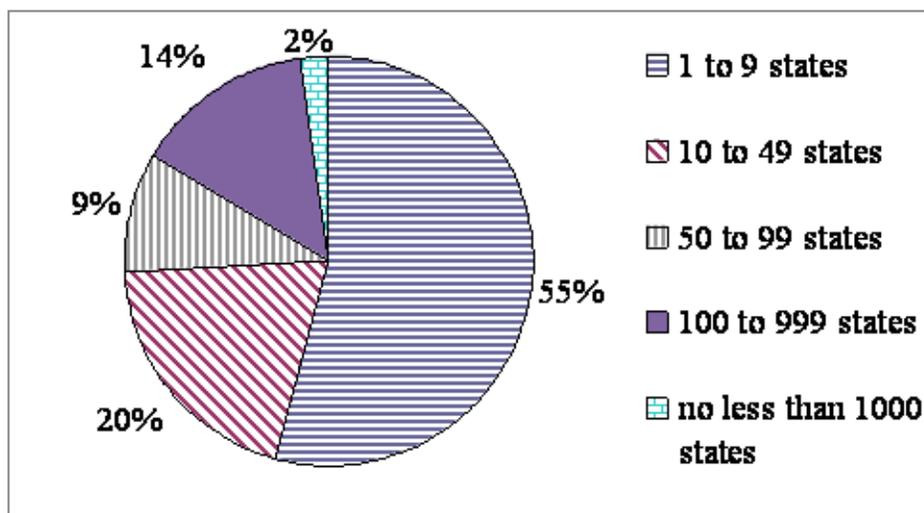


Figure 5.6: Distribution of the number of unique functional states (*design_A*).

PRBS generator to generate patterns for the inputs of *design_A*. Also, we extract and apply functional constraints during functional scan test.

Figures 5.6-5.8 highlight the functional state space information for *design_A*. The distribution of the number of unique functional states, unique functional state transitions, and unique functional state sequences of depth 3 under the given functional test are shown in these figures, respectively. For example, from Figure 5.6 we can see that 55% substate groups traverse 1-9 unique functional states under the given functional test. These figures also show that most of the substate groups traverse only a small number of unique functional states, unique functional state transitions and unique functional state sequences of depth 3. For example, although the given functional test includes 29,317 clock cycles, Figure 5.6 shows that only 2% substate groups traverse 1,000 or more unique functional states.

Table 5.2 lists the average values of α -coverage metrics on all groups for functional scan test of *design_A*. Column 1 shows the various value for parameter α . Columns 2-4 list the average values of α -coverage metrics when functional constraints are not applied. Columns 5-7 refer to the case when functional constraints are applied. From

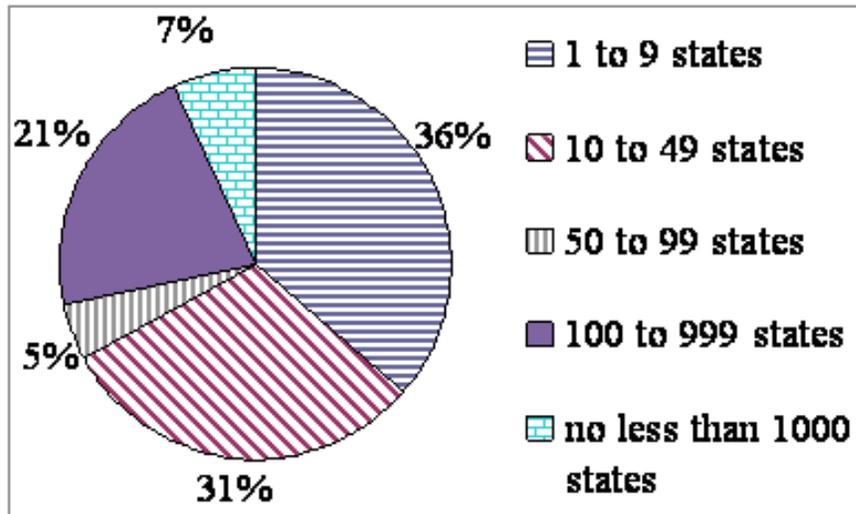


Figure 5.7: Distribution of the number of unique functional state transitions (*design_A*).

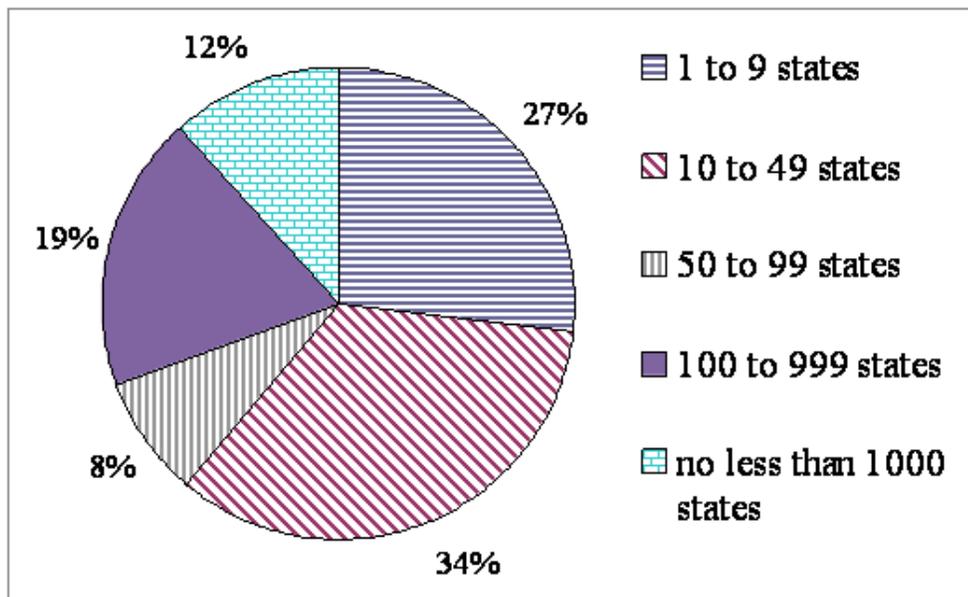


Figure 5.8: Distribution of the number of unique functional state sequences of depth 3 (*design_A*).

Table 5.2: α -coverage metrics for functional scan test for *design_A*.

α	No constraints			With constraints		
	Cov_{S_α}	Cov_{ST_α}	$Cov_{SS_{k_\alpha}}$ ($k = 3$)	Cov_{S_α}	Cov_{ST_α}	$Cov_{SS_{k_\alpha}}$ ($k = 3$)
0.8	86%	85%	84%	91%	89%	87%
0.85	81%	79%	78%	87%	84%	81%
0.9	74%	72%	70%	81%	77%	72%

this table, we can see that higher the value of α , lower the value of the α -coverage metrics. For example, with no functional constraints, we obtain 86% for Cov_{S_α} when α is set to 0.8, but only 81% when α is set to 0.85. This is because higher α means we need more similarity between a functional state and a functional scan state, an objective that is more difficult to achieve; thus we get a lower value for α -coverage. Therefore, a balance is needed between the values of α and the α -coverage. This table also shows that by applying appropriate functional constraints during functional scan test, we can increase values for the α -coverage metrics for the same value of α . We observe that by applying functional constraints during functional scan test, we obtain reasonably high values for α -coverage metrics with high α . The above results indicate that the functional state space can be mimicked well, thus it is possible to use functional scan test (with proper functional constraints) to reproduce and detect functional failures for *design_A*.

5.5.3 Reproducing and detecting functional failures

In this section, we use the flow in Section 5.4 to reproduce and detect functional failures in functional scan test. In the reproduction and detection procedure, for *design_A*, we divide the functional test into 3 stages. Then we find the first failing stage. Next, we obtain the initial cycle and dump the initial state in functional test.

We start from the initial state and apply functional scan test (values of PIs come from the PRBS generator and the extracted functional constraints). Finally, we check whether the functional failure is reproduced and detected in functional scan test.

We first artificially create several faulty boards. Each faulty board is obtained by injecting one stuck-at fault, dominant bridging fault, crosstalk fault or a delay fault on a fault-free board. Even though high stuck-at coverage is typically obtained for chip testing, we consider stuck-at faults as a simple approach to model additional board-level failures. Crosstalk usually results from capacitive coupling between wires [101]. We randomly generate aggressor/victim pairs in this chapter. When the aggressor and victim nets fall or rise simultaneously and in opposite directions, a delay is injected to the falling or rising of the victim net.

We also consider delay faults on non-critical paths that are not targeted for chip testing but might cause fails at board level due to power-supply noise on the board. Power-supply noise is considered as one of the major causes for board-level functional failures [86]. The ATE test environment and board-level functional test environment tend to differ in terms of power supply and clock characteristics. In ATE test, only a few clock cycles (typically one or two clock cycles) are considered for response capture. On the other hand, board-level functional test involves millions of clock cycles, which is close to operation in functional mode. Therefore, a board-level functional test can often detect delay faults due to power-supply noise that are missed by the ATE during chip testing. We simulate board-level functional failures due to delay faults that escape chip-level testing. Since delays on critical paths or paths with small slack are targeted in chip testing, we only consider delay faults on shorter paths that might become critical on the board due to noise. We target the scenario that board-level noise results in additional delays on non-critical paths if a large fraction of signals in the design have high switching activities (we assume that at least 90%

Table 5.3: Functional failure detection using functional scan test for *design_A*.

Fault model	No. of functional failures	No. of detected faults in functional scan test	Percentage of success
stuck-at	100	100	100%
bri-dom	100	100	100%
crosstalk	65	54	83.08%
delay	100	78	78%

of the signals toggle in one clock cycle) within a given window of clock cycles (10 consecutive cycles in our work). We create a faulty board by injecting a delay on 10 short paths (identified by static timing analysis) during test execution when the switching activities of signals meet the condition described above.

Table 5.3 list the number of functional failures detected by using functional scan test for *design_A*, respectively. Column 2 shows the number of functional failures due to each type of fault. Column 3 shows the number of faults detected in functional scan test. Column 4 shows the percentage of detected functional failures, relative to the total functional failures. We find that for *design_A*, functional scan test can detect most of the functional failures due to all four types of injected faults. In particular, all the 100 functional failures each due to stuck-at fault and bridging fault can be detected using functional scan test.

5.5.4 Functional failures due to more random faults

In Section 5.5.3, we have shown the percentage of reproduced and detected functional failures using functional scan test for about 100 faulty boards due to each type of fault. In order to further verify the effectiveness of the proposed method, we perform experiments for functional failures due to more random faults of each type. We ran multiple experiments, corresponding to 100, 200, 300, . . . , 1,000 faults, respectively, for four fault models (stuck-at, bridging, crosstalk, and delay caused by power-supply-

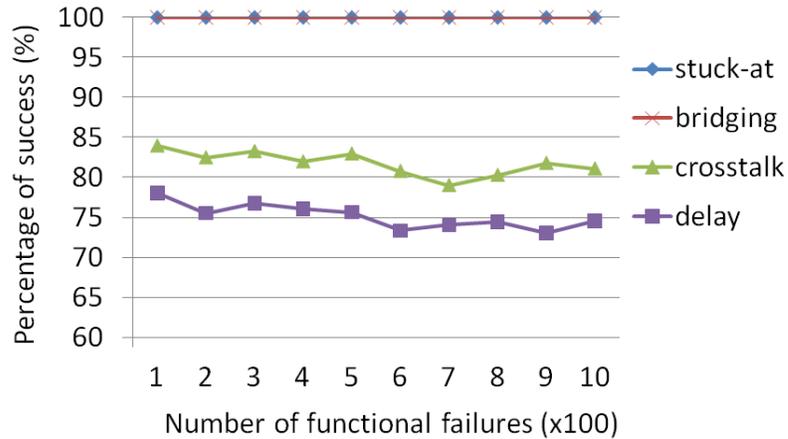


Figure 5.9: Reproduction and detection results for functional failures due to more random faults.

noise). Each artificial faulty board is obtained by injecting one fault. Figure 5.9 shows the reproduction and detection results using functional scan test for these functional failures. We can see that we can consistently obtain 100% success (percentage of functional failures that can be reproduced by functional scan test) for functional failures due to stuck-at faults and bridging faults. For crosstalk faults and delay faults due to power supply noise, the success percentage varies but it is still high.

5.6 Automated Extraction on Functional Constraints

A key challenge in the proposed method is how to vary values of PIs for the target component so that we can better mimic the functional state space, thus reproducing and detecting functional failures effectively using functional scan test. In the methodology flow and experimental results discussed in Section 5.5, the values for data signals come from the PRBS generator and the values for control signals are obtained by manually extracting constraints from RTL design and test sequences. In this section, we automate the flow to extract functional constraints as well as improve the extracted functional constraints.

For data signals, we use the weighted PRBS. That is, we consider the 1-probability on corresponding inputs. For example, suppose the given functional test is 100 clock cycles in length and a data input *data1* is set to 1 in 30 clock cycles. Then the 1-probability of *data1* is 0.3 under the given functional test. Therefore, we adjust the PRBS generator to generate data with 1-probability of 0.3 for *data1*, which is used in functional scan test.

For control signals, we first simulate the original functional test at RT-level to monitor the values of targeting control signals. Next we use an in-house tool to automatically extract constraints. Currently, the tool can extract constraints according to the following five rules:

Rule 1: Signals with constant value. For example, *sig1* is always high during the execution of the functional test;

Rule 2: Signals that is periodically repeatable. For example, *sig1* is set to high every 10 clock cycles.

Rule 3: Signals that should last for several cycles. For example, *sig1* remains its value for at least 4 cycles once it is asserted.

Rule 4: Correlated signal pair. For example, *sig1* always has the same value as *sig2*; *sig3* always has the opposite value of *sig4*.

Rule 5: Signal pair that has temporal relationship. For example, *sig1* is always asserted one cycle after *sig2* is asserted.

By automatically extracting the improved functional constraints as above and apply them during the running of functional scan test, we can enhance the mimicking of functional state space as well as increase the success percentage of reproduction and detection of functional failures. Table 5.4 shows the α -coverage metrics for functional scan test with enhanced functional constraints. We can see that compared to manually extracted functional constraints, the enhanced functional constraints can

Table 5.4: α -coverage metrics for functional scan test with improved functional constraints.

α	With constraints			With enhanced constraints		
	Cov_{S_α}	Cov_{ST_α}	$Cov_{SS_{k_\alpha}}$ ($k = 3$)	Cov_{S_α}	Cov_{ST_α}	$Cov_{SS_{k_\alpha}}$ ($k = 3$)
0.8	91%	89%	87%	93%	91%	88%
0.85	87%	84%	81%	90%	86%	82%
0.9	81%	77%	72%	84%	80%	74%

Table 5.5: Functional failure detection using functional scan test with improved functional constraints.

Fault model	No. of functional failures	Percentage of success	
		with constraints	enhanced constraints
stuck-at	100	100%	100%
bri-dom	100	100%	100%
crosstalk	65	83.08%	87.70%
delay	100	78%	81%

better mimic the functional state space in terms of α -coverage metrics. Moreover, the percentage success in the reproduction and detection of functional failures is also increased (Table 5.5).

5.7 Techniques to Mimic Functional State Space

All the experimental results presented above show that we can effectively reproduce and detect functional failures using functional scan test if we can mimic functional state space well. In this section, two techniques are proposed to adequately mimic the functional state space using functional scan tests. The first technique is based on the selection and use of multiple initial states in functional scan tests. The other approach is to adopt state-injection techniques to accompany functional scan tests.

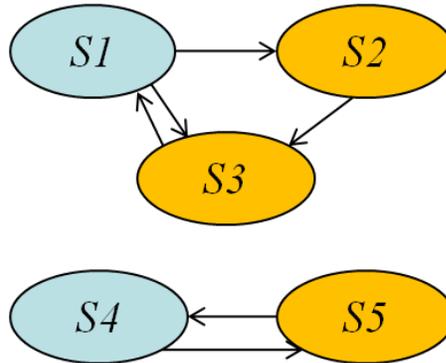


Figure 5.10: Motivation for the use of multiple initial states.

5.7.1 Multiple initial states

Let us use Figure 5.10 as an example to explain our motivation for selecting multiple initial states. Suppose that from the specification of a design, the state transition graph is extracted as shown in Figure 5.10. Each node represents a functional state and each directed edge represents a possible state transition. For example, an edge from S1 to S2 implies that there is a possible state transition from S1 to S2. If we start from one initial state S1 and run a test for many capture cycles, the resulting functional scan test may only traverse three states S1, S2 and S3. Similarly, if the functional scan test starts from initial state S4, it may only traverse two states S4 and S5. In this case, multiple initial states are necessary with functional scan tests in order to adequately mimic the functional state space.

Given a number t , it is non-trivial to select t initial states such that the derived functional scan tests can effectively mimic functional state space. Consider Figure 5.10 as an example. If we are allowed to select two initial states, S1 and S4 are better candidates than S1 and S2 since starting from S1 and S4 leads us to higher state space coverage. Next, we propose an effective approach for the selection of initial states based on the connectivity property of a directed graph.

The state transition graph obtained from a particular design and the given func-

tional test is considered as a directed graph, referred to as G . To facilitate the selection of initial states, we assign each node (state) a weight (see Figure 5.11). The weight for node i is defined as $C_i - C_{ii}$, where C_i is the number of cycles for which the state i is visited under the given functional test, and C_{ii} is the number of cycles for which the state i is visited and the circuit state in the previous cycle is also state i . A directed graph is called *strongly connected* if there exists a path from each of its vertices to every other vertex [102]. The *strongly connected components (SCC)* of a directed graph G are its maximal strongly connected subgraphs. If each strongly connected component is contracted to a single vertex, the resulting graph G' is a directed acyclic graph (DAG), i.e., the *condensation* of G . It is known that there is at least one source node of indegree 0 in a DAG [103]. The proposed method for initial-states selection is summarized below, followed by a detailed description.

1. Step 1: Generate the weighted state transition graph based on the given functional test sequence and the design;
2. Step 2: Obtain the *SCCs* of the graph in Step 1 and its corresponding DAG G' ;
3. Step 3: Starting from the source node of DAG, we order the nodes in G' using a breadth-first search algorithm, which is equivalent to order the *SCCs* in G .
4. Step 4: *SCCs* take turns in contributing their states with the lowest weight as the initial state and the selected states are removed from the *SCC*¹;
5. Step 5: Step 4 continues until the number of selected initial states reaches t ;

In Step 1, we generate the weighted state transition graph by running Verilog simulation using the given design and the given functional test sequence. An example of weighted state transition graph is given in Figure 5.11; it is composed of

¹The first *SCC* will follow the last one if t is larger than the number of *SCCs*.

12 functional states, where each node represents a functional state. The circuit has state transitions from state 2 and state 6 to state 3 for 10 clock cycles, therefore the weight for node 3 is 10.

In Step 2, we search the *SCCs* over the weighted state transition graph and generate the DAG. A discussion of the search algorithms for the *SCCs* is not included here; details are available in [103]. The computational complexity of the algorithm is $O(|V| + |E|)$, where $|V|$ and $|E|$ refer the numbers of vertices and edges in graph G , respectively. The number of vertices $|V|$ in the graph G equals the number of unique states reached by the given functional test; it is upper-bounded by the length of this test. Note that $|E| = O(|V|^2)$.

For the graph in Figure 5.11, we can decompose it into four *SCCs* shown as yellow dotted rectangles. The first *SCC* is composed of state 1; the second *SCC* is composed of state 2, state 4 and state 5; and the third *SCC* is composed of state 3 and state 6. The fourth *SCC* is composed of the remaining states. The condensed DAG G' for these *SCCs* is shown in Figure 5.12. Each node in G' represents a *SCC*. Each edge in G' indicates that there is a path from at least one state in one *SCC* to a state in another *SCC*.

In Step 3, we order the nodes in G' using the breadth-first search algorithm, i.e., order the *SCCs* in G . This ordering is of considerable significance. For example, for the DAG in Figure 5.12, the ordering obtained for *SCCs* is: the *SCC* 1, *SCC* 2, *SCC* 3, and *SCC* 4. By starting from states in the *SCC* 1, the design may also traverse states in the *SCC* 2 after many cycles. While if starting from states in the *SCC* 2, the design will never traverse states in the *SCC* 1. This property can be incorporated in the selection of initial states.

In Steps 3–5, we select t initial states based on the DAGs generated and the weight information for each state. For example, given $t = 5$, we select five initial states as

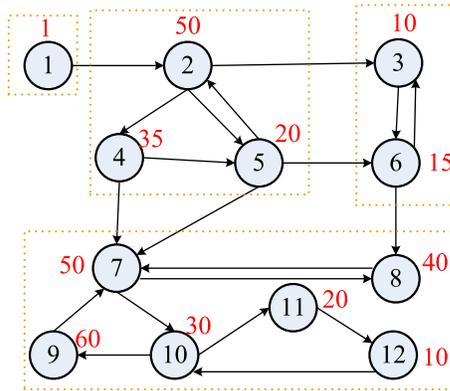


Figure 5.11: An example of weighted state transition graph G .

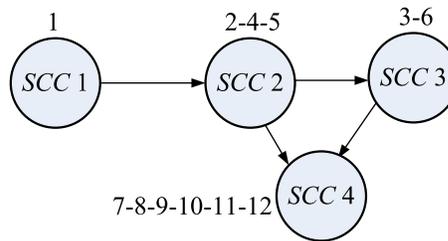


Figure 5.12: Graph G' for strong connectivity components.

follows: beginning from the source node of the DAG in Figure 5.12, we select state 1 as the first initial state since only state 1 belongs to this SCC . Next, we turn to the second SCC that includes states 2, 4 and 5. Since state 5 has the lowest weight (intuitively speaking, the state with the lowest weight is the one that is the hardest to visit), we select state 5 as the second initial state. Following this procedure, we select states 3 and 12 as the third and fourth initial states, respectively. Since t is larger than the number of $SCCs$ (4), we go back to the first SCC after the fourth initial state is chosen. Since no other state is left in the first SCC , we go to the second SCC and select state 4 as the fifth initial state. In this way, we select five initial states and they are expected to lead to functional scan tests that can effectively mimic the functional state-space of the given functional test.

5.7.2 State-injection technique

By using multiple initial states, we can use functional scan tests to effectively mimic functional state space. However, there is a time penalty associated with this method, since we have to load each initial state using slow clock signals. Furthermore, there may exist some states, state transitions, and sequence of states that cannot be covered even if we adopt multiple initial states. An alternative method is to use a state-injection technique during the capture procedure of functional scan test. Starting from an initial functional state S_0 , suppose that the system reaches state S_n after we have run a given number of capture cycles in functional scan test mode. The state S_n can be at large Hamming distance from a state in the functional state space traversed by the given functional test. In this case, we identify S_n as being ineffective for our purpose, and modify the current state S_n to a target functional state and continue running capture cycles based on the modified system state.

We propose a method to combine the state-injection technique with functional scan test to mimic the functional state space. It is based on the α -coverage of functional state sequence of depth k . Details are presented below.

- Step 1: Obtain the set of functional states traversed by the given functional test. Label it as the set of target functional states S_{tar} . For each state in S_{tar} , obtain and record the functional state sequence of depth k starting from this state.
- Step 2: Start from the given initial functional state S_0 to run functional scan test.
- Step 3: Run m capture cycles of functional scan test (m is a user-defined parameter). Let the current system state be S_{cur} ;

- Step 4: Calculate α -coverage of functional state sequence of depth k for each state in S_{tar} . Find the state that has the lowest coverage value and record it as S_{cand} . Remove those states from S_{tar} whose values are higher than the given threshold α .
- Step 5: Check whether state S_{cur} is an ineffective state, i.e., the Hamming distance of S_{cur} to every state in S_{tar} is higher than a threshold. If yes, modify the current state to be S_{cand} by state-injection logic and go to Step 3. If not, go to Step 3.

In Step 3, we check how the functional state space is being mimicked, and perform state injection periodically as needed. For example, m may be set to 100. That is, after every 100 capture cycles of functional scan test, we check whether there is need to inject a new state. In Step 4, for those states whose state sequences of depth k have been adequately mimicked, we remove them from the set of target functional states and do not need to consider them as candidates in the subsequent state-injection step. In Step 5, if the current state is too far from each of the remaining target functional states, it is an ineffective state and we perform state injection to alter the current state to S_{cand} .

Computational Complexity: The computational complexity of Step 1 is $O(|S_{tar}| \cdot k)$, where $|S_{tar}|$ is the size of S_{tar} , representing the number of unique functional states, and k is the depth of functional state sequences. For Step 3, the computational complexity is $O(m)$, where m is the time interval, in terms of the number of cycles, before checking the necessity to inject a new state. For Step 4, the computational complexity is $O(n(\text{func_state_seq_}k) \cdot |S_{tar}|)$, where $n(\text{func_state_seq_}k)$ is the number of unique functional state sequences of depth k and is upper-bounded by $\binom{|S_{tar}|}{k}$. Since an upper bound on $\binom{|S_{tar}|}{k}$ is $|S_{tar}|^k$, the computational complexity of Step 4 is thus $O(|S_{tar}|^{k+1})$. For Step 5, the computational complexity is $O(|S_{tar}|)$. Suppose

that the maximal number of capture cycles allowed for functional scan test is C_{max} . Based on the above analysis, the computational complexity \mathcal{C} for the state injection procedure is

$$\begin{aligned}
\mathcal{C} &= O(|S_{tar} \cdot k| + m + |S_{tar}|^{k+1} + |S_{tar}|) \cdot \frac{C_{max}}{m} \\
&= O(m + |S_{tar}|^{k+1}) \cdot \frac{C_{max}}{m} \\
&= O(C_{max}) + O(|S_{tar}|^{k+1} \cdot \frac{C_{max}}{m})
\end{aligned} \tag{5.6}$$

Since $|S_{tar}|^{k+1}/m \geq 1$, (5.6) can be simplified as

$$O(|S_{tar}|^{k+1} \cdot \frac{C_{max}}{m}), \tag{5.7}$$

which means that the computational complexity of the state injection procedure increases with the number of unique functional states ($|S_{tar}|$), the depth of the targeted functional state sequences (k), and the maximal number of capture cycles allowed for the functional scan test (C_{max}). The complexity decreases with m .

The hardware implementation of the state-injection technique is similar to that of fault insertion (FI), as described in [104]. The difference lies in the fact that the fault insertion technique intentionally inserts wrong values at pin/logic level to model the effects of manufacturing defects while the state-injection technique injects specific values to flip-flops to modify the current circuit state to a target state. However, it is potentially expensive to enable state injection at all the flip-flops in a large design. Also, in order to periodically check whether we need to inject a new state and determine which target state must be injected, we need to store the information on traversed state (sequences) up to the current capture cycle of the functional scan

test. In other words, the state injection technique removes the need of additional time needed for loading multiple initial states, but is associated with potentially high hardware overhead.

5.7.3 Experimental results

Experiments were performed using *design_A* to first show that by selecting and using multiple initial states, functional scan test with proper functional constraints can adequately mimic functional state space, measured by Cov_S_α , Cov_ST_α , and $Cov_SS_k_\alpha$. We also show that the state-injection technique combined with functional scan test can lead to steeper cumulative $Cov_SS_k_\alpha$ curve. The experimental setup is similar as in Section 5.5.1.

Simulation results for multiple initial states

For *design_A*, we consider 63 substate groups that are related to the given functional test. The functional test for *design_A* includes 29,317 clock cycles and the functional scan test starting from one initial state is set to have 10,000 capture cycles.

First, for each substate group, we record the number of unique functional states, state transitions, and state sequences traversed by the given functional test. Next, for the given value of t , we use the method described in Section 5.7.1 to select t initial states for each substate group and compose t global initial states. For example, the first global initial state is composed of the first initial states for each substate group. Following this, we apply functional scan test starting from the obtained t global initial states and calculate the α -coverage for functional states, state transitions, and state sequences. Finally, we observe whether we can obtain high values of α -coverage metrics with a large value of α . We also compare the results with the method of randomly selecting t initial functional states.

Since we obtain higher values for α -coverage metrics with the application of functional constraints, we adopt the scheme of applying extracted functional constraints to functional scan test in the following initial-state selection experiments. We first obtain 5 initial states for *design_A* using our proposed method (described in Section 5.7.1) and the random method, respectively. Next, α -coverage metrics for each substate group are calculated for the resulting functional scan tests (with applying functional constraints), starting from the selected 5 initial states by two methods. Table 5.6 compares the average values of α -coverage metrics on all groups for resulting functional scan tests of *design_A*, starting from the 5 initial states selected by our proposed method and the random method. The depth of functional state sequence (k) is set to 3. The number of capture clock cycles is set to 10,000 for the functional scan test starting from each initial state. We can see that, by using the selected initial states with our proposed method, the values of α -coverage metrics can be enhanced comparing with the random method. For example, as shown in Table 5.6, the Cov_S_α for $\alpha = 0.8$ has been increased from 87.96% to 90.16% comparing with the random method. We can see that our proposed initial-state selection method is more effective than the random selection method in the terms of α -coverage metrics. The above results indicate that for *design_A*, we can use functional scan test (with proper functional constraints) to mimic the functional state space well by selecting and using 5 initial states, thus it is possible to use functional scan test to reproduce and detect functional failures for *design_A*.

Simulation results for the state-injection technique

We next examine the effectiveness of the proposed state-injection approach described in Section 5.7.2. The procedure presented in Section 5.7.2 is based on the α -coverage of functional state sequence of depth k . Without loss of generality, we set k to 3

Table 5.6: Comparison of α -coverage metrics for the proposed initial-state selection method and random method for *design_A*.

α	5 random initial states			5 selected initial states with proposed method		
	Cov_{S_α}	Cov_{ST_α}	$Cov_{SS_{k_\alpha(k=3)}}$	Cov_{S_α}	Cov_{ST_α}	$Cov_{SS_{k_\alpha(k=3)}}$
0.8	87.96%	86.97%	85.78%	90.16%	89.53%	88.49%
0.85	83.24%	80.85%	79.28%	85.29%	83.67%	82.17%
0.9	76.46%	73.90%	69.44%	78.19%	75.92%	72.12%

in the following state-injection experiments. That is, for a functional state S_i , if we have already obtained high values on the α -coverage of functional state sequences (starting from S_i) of depth 3, we will no longer consider S_i as a candidate state for state injection.

Figures 5.13-5.15 show the state-injection results for *design_A* for various value of α . For *design_A*, we start from one randomly chosen functional state and run 10 passes of the state-injection procedure. In each pass, we run 1,000 capture cycles of the functional scan test. At the end of each pass, we update the α -coverage (α is set to 0.8, 0.85, and 0.9 respectively) for each state in S_{tar} , select the candidate functional state that has the lowest value in α -coverage, inject the selected state and continue running the capture procedure of the functional scan test. Figures 5.13-5.15 compare the cumulative $Cov_{SS_{k_\alpha(k=3)}}$ for two cases of with and without state-injection for *design_A*. When state injection is not used, we simply start from the initial functional state and run 10,000 capture cycles of the functional scan test. After every 1,000 capture cycles, we calculate the $Cov_{SS_{k_\alpha(k=3)}}$ and compare with the case when state injection is used. From Figures 5.13-5.15, we can see that the state-injection based method leads to much steeper ramp-up and cumulative α -coverage curve. This implies that functional scan test combined with state injection is more effective in mimicking the functional state space.

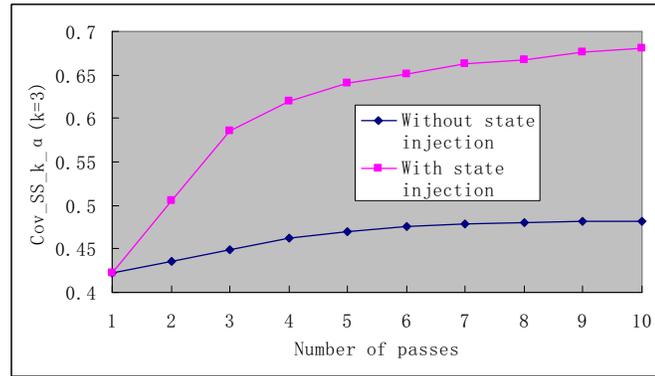


Figure 5.13: Comparison of $Cov_{SS_k_\alpha}$ ($k = 3, \alpha = 0.8$) for with and without state-injection (*design_A*).

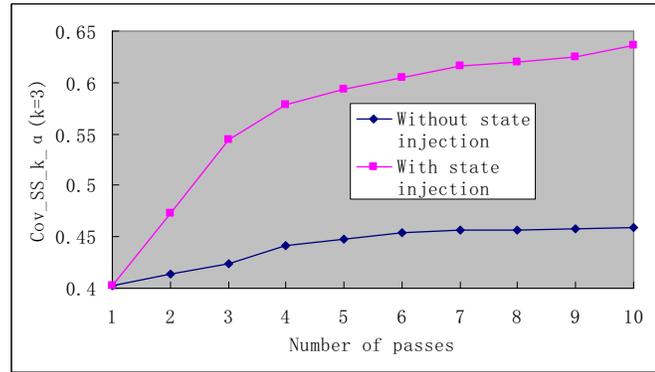


Figure 5.14: Comparison of $Cov_{SS_k_\alpha}$ ($k = 3, \alpha = 0.85$) for with and without state-injection (*design_A*).

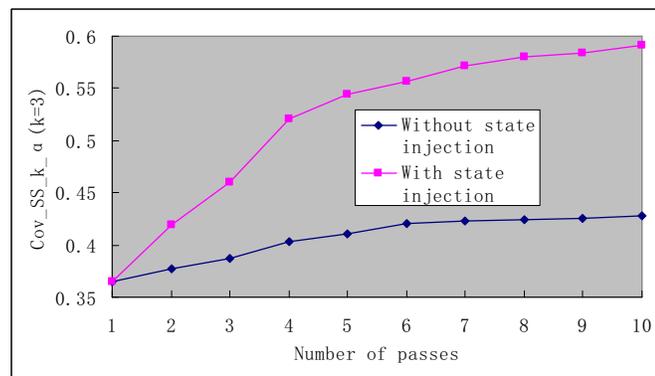


Figure 5.15: Comparison of $Cov_{SS_k_\alpha}$ ($k = 3, \alpha = 0.9$) for with and without state-injection (*design_A*).

Compared to the method of choosing multiple initial states, state injection provides better mimicking of the functional state space, but it is associated with higher overhead, measured in terms of storage requirement (for storing states or state sequences traversed by the functional scan test) and extra hardware logic (to make all the flip-flops ready for state injection).

5.8 Chapter Summary and Conclusions

In this chapter, we have proposed a DFT technique for analyzing board-level functional failures. We have also presented a new method for using functional scan test to reproduce and detect board-level functional failures. The DFT technique and the associated reproducing and detection flow allow us to reproduce and detect functional failures in a controlled deterministic environment, following which we can diagnose the root cause of a board-level functional failures and obtain tests that can be ported to ATE for defect screening. Experiments for an industry design first show that functional scan test with appropriate functional constraints can mimic the functional state space adequately, thus it is feasible to use functional scan test to reproduce and detect functional failures. Experiments also show that most functional failures due to stuck-at, bridging, crosstalk and delay faults due to power supply noise can be detected using functional scan test with appropriate functional constraints. We have also presented two approaches for using functional scan test to better mimic functional state space in order to obtain higher success percentage of reproduction and detection of functional failures. The first approach focuses on the selection and use of multiple initial states. The second approach adopts the state-injection technique. Experiments show that our proposed initial-state selection method and state-injection method can allow functional scan test with appropriate functional constraints to better mimic the functional state space.

Chapter 6

Conclusions and Future Work

Test data compression offers a promising solution to reduce test data volume and test application time. LFSR reseeding based test compression is one of the most common techniques. In this technique, a larger number of seeds can be computed by solving linear equations based on the given polynomial and test cube. Since different seeds lead to patterns with various qualities, there is a need to select the effective seeds that can derive high-quality patterns. The research reported in this dissertation presents a seed-selection technique that uses the method of output deviations for targeting unmodeled defects. In order to further reduce the memory requirement, a seed augmentation method based on bit-operations are also reported in this thesis.

Functional tests, usually comes from design verification tests, are commonly used in industry. Since it is impractical to apply them all in the manufacturing testing, it is of special interest to evaluate the quality of functional tests and only apply the effective ones. The traditional test grading is usually performed by gate-level fault simulation. However, it is too time-consuming. In this thesis, we have shown how to grade functional test sequences at RT-level based on RT-level output deviations. In this way, we can avoid the time-consuming fault simulation. Instead, only RT-level logic simulation is needed. Functional test however suffers from low defect coverage since it is often the case that the observability is not considered when they are developed for design verification. In this thesis, we have also described an observation-point selection and insertion method to improve the quality of given functional tests.

NTF is a common scenario in system companies and it is hard to find the root cause of NTF when it is caused by board-level functional failures. In this thesis, we

propose the DFT techniques and diagnosis methods to reproduce and detect board-level functional failures in a controlled and deterministic environment, thus facilitate the subsequent more fine-grained diagnosis procedure. In this thesis, we also propose a new method to rank candidate faulty blocks based on dataflow analysis and DS theory for the diagnosis of board-level functional failures.

This thesis has presented a multi-pronged approach to address the above issues. The results of this research have led to several useful techniques that can be used to reduce manufacturing test and diagnosis cost.

6.1 Thesis Contributions

Chapter 2 presented the seed selection method based on output deviations for LFSR-reseeding-based test compression. The goal is to select seeds that are expanded to patterns with high output deviations. Experimental results show that the selected seeds lead to high-deviation patterns, which provide higher defect coverage than patterns derived from other seeds. Moreover, faster fault-coverage ramp-up is obtained using these LFSR seeds. We have also presented a DFT method for increasing the effectiveness of LFSR reseeding. We have shown how on-chip augmentation of LFSR seeds can be combined with output deviations to ensure that the most effective patterns are applied to the circuit under test. Simulation results have demonstrated that for the same pattern count, the proposed method utilizes significantly fewer seeds, yet provides nearly the same defect coverage ramp-up as a seed-selection method based only on output deviations. For the same number of seeds, the proposed method provides higher coverage and faster coverage ramp-up compared to the method without seed augmentation.

Chapter 3 presented the output deviation metric at RT-level to model the quality of functional test sequences. By adopting the deviation metric, timing-consuming

fault simulation at gate-level can be avoided for the grading of functional test sequences. Experiments show that the obtained deviations at RT-level correlate well with the stuck-at (transition / bridge) fault coverage at gate-level. Moreover, the reordered functional test sequences set based on deviation method has a steeper cumulative gate-level fault coverage curve. This is significant in the mass production. We have also presented a method to select and insert the observation points for a given RT-level design and a functional test sequence. This DFT approach allows us to increase the effectiveness of functional test sequences (derived for pre-silicon validation) for manufacturing testing. Experiments show that the proposed RT-level DFT method outperforms two baseline methods for enhancing defect coverage. We also show that the RT-level deviations metric allows us to select a small set of the most effective observation points.

Chapter 4 presented a new method to rank candidate faulty blocks based on dataflow analysis and DS theory for the diagnosis of board-level functional failures. The proposed method transforms the information of a functional test failure into the information of multiple-stage failures by partitioning the given functional test into multiple stages. A belief measure is assigned to each block based on each failing stage and DS theory is used to combine the beliefs from multiple failing stages. Experiments for an industry design show that the proposed method provides effective rankings for most of the board-level functional failures due to injected faults.

Chapter 5 presented a DFT technique for analyzing board-level functional failures. We have also presented a new method for using functional scan test to reproduce and detect board-level functional failures. The DFT technique and the associated reproducing and detection flow allow us to reproduce and detect functional failures in a controlled deterministic environment, following which we can diagnose the root cause of a board-level functional failures and obtain tests that can be ported to ATE

for defect screening. Experiments for an industry design first show that functional scan test with appropriate functional constraints can mimic the functional state space adequately, thus it is feasible to use functional scan test to reproduce and detect functional failures. Experiments also show that most functional failures due to stuck-at, bridging, crosstalk and delay faults due to power supply noise can be detected using functional scan test with appropriate functional constraints. We have also presented two approaches for using functional scan test to better mimic functional state space in order to obtain higher success percentage of reproduction and detection of functional failures.

6.2 Future Work

This thesis gives rise to a number of exciting research directions. Here we summarize three extensions to the thesis research that are relevant for DFT and diagnosis methods.

6.2.1 Diagnosis Based on a Fault Model

In the thesis, we have used functional scan test to reproduce and detect the board-level functional failures. Following this, we can diagnose the root cause of a board-level functional failure to specific wires/gates inside a component based on the detection information and given fault models.

Let us use Figure 6.1 to explain this. The golden board is abbreviated as GLD and the failing board is abbreviated as CUD. The reproduction and detection of a board-level functional failure includes three major steps. In the first step, the given functional test is partitioned to several stages and at the meanwhile it is applied on the GLD and CUD in functional test mode to get the failure syndrome. In the second step, we apply functional test on GLD and CUD in functional scan mode.

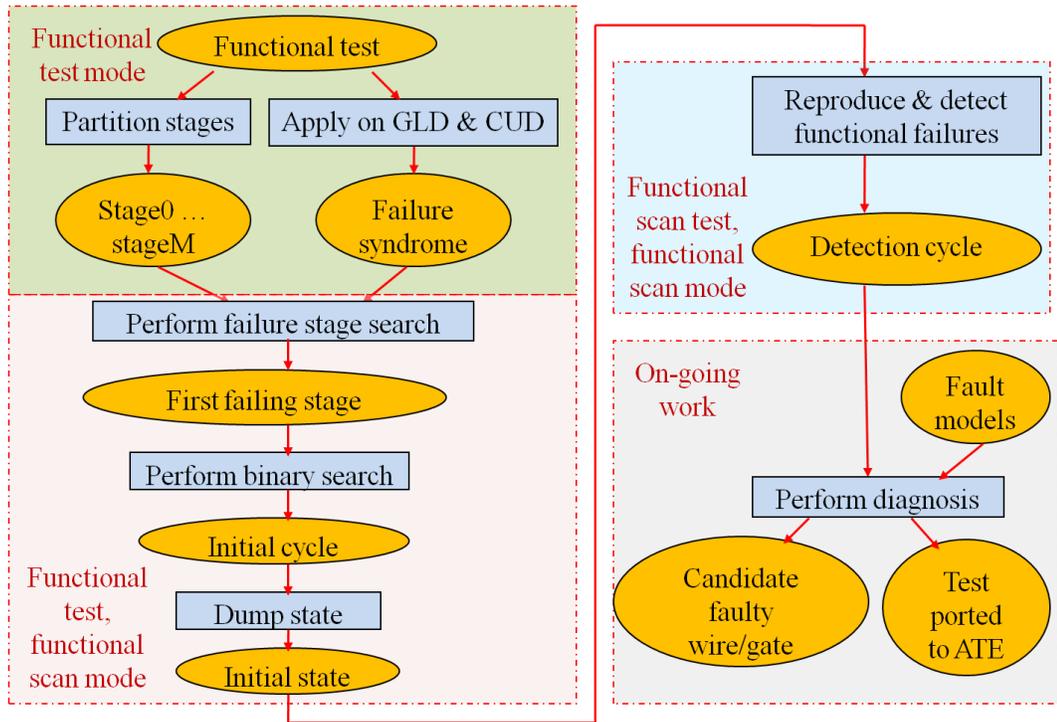


Figure 6.1: The diagnosis flow of functional failures.

The purpose of this step is to obtain the initial state, which is necessary to trigger the failure. In the third step, the boards operate in the pure functional scan mode. In this step, functional scan test with appropriate constraints is used to reproduce and detect the functional failures and the detection information is recorded. The above is what we have done in the thesis work.

A possible direction would be the development of algorithms and methods to perform diagnosis based on target fault models and the obtained detection information. The goal is to root cause the functional failure to specific faulty wires/gates inside a component as well as obtain the tests that can be ported to ATE.

6.2.2 Additional Diagnostic Test Generation

Another research direction would be developing methods to perform constrained ATPG to generate new diagnostic tests and develop algorithms to validate the can-

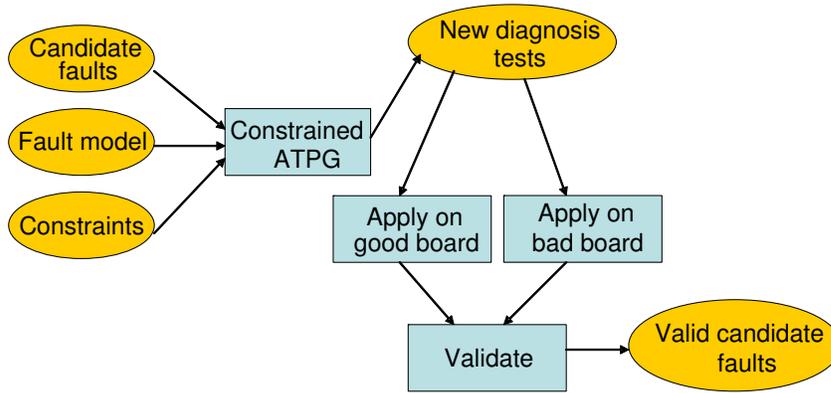


Figure 6.2: Flow for constrained ATPG and validation.

didate faults efficiently.

After obtaining candidate faults for the board-level functional failures, we can generate additional diagnosis tests via constrained ATPG. The new tests thus generated can be used to validate the candidate faults and further improve the accuracy of diagnosis. The flow for constrained ATPG and validation is shown in Figure 6.2. New diagnosis tests can be generated for the candidate faults based on the fault model and constraints. These new tests can then be applied on a good board and a faulty board separately to further validate the identification of candidate faults.

Figure 6.3 shows an example of validation using the new diagnosis tests. Suppose the candidate faults are $f1$, $f2$ and $f3$ (shown in Column 1 in the table). Suppose that for each fault, three additional diagnostic tests are generated. For example, $p1$, $p2$ and $p3$ are new diagnostic tests for $f1$. Suppose these tests are applied to a good board and a faulty board. If the state values and POs are all the same for both cases, we denote the test as pass. Otherwise, denote it as fail. The pass/fail information of each new test is shown in the last column of the table. If a fault is a root cause of the functional failure, the new tests targeting it should all fail. In this example, $f1$ and $f3$ is not the root cause. So the candidate fault set has been narrowed down to one fault, namely $f2$.

	p1	p2	p3	p4	p5	p6	p7	p8	p9
f1	✓	✓	✓						
f2				✓	✓	✓			
f3							✓	✓	✓
pass /fail	pass	fail	fail	fail	fail	fail	pass	pass	fail

Figure 6.3: Validation example.

6.2.3 Fault Diagnosis for Designs with Communication to Memory

A memory block is usually bypassed in traditional ATPG mode. When the design communicates with memory, diagnosis is complicated since it needs to run exhaustive deterministic tests with the appropriate memory contents. In order to solve this problem, one method is to download the contents of memory in the functional mode and cycle through them to the related interface in the deterministic tests mode. Another method is to design logic to make the memory un-bypassed in the deterministic test mode. The initial values of the memory in the deterministic test mode are loaded from the values downloaded in functional mode. Based on the above problem statement, a promising research direction would be developing DFT techniques and algorithms to perform diagnosis for designs that communicate with memories.

Bibliography

- [1] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*, IEEE Press, Piscataway, NJ, 1990.
- [2] M. L. Bushnell and V. D. Agrawal, *Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits*, Boston; Kluwer academic publishers, 2000.
- [3] M. Sachdev, *Defect Oriented Testing for CMOS Analog and Digital Circuits*, Boston; Kluwer Academic Publishers, 1998.
- [4] R. A. Rutman, "Fault detection test generation for sequential logic heuristic tree search," *IEEE Computer Repository Paper*, no. R-72-187, 1972.
- [5] L. H. Goldstein, "Controllability/observability analysis of digital circuits," *IEEE Trans. Circuits and Systems*, vol. CAS-26, no. 9, pp. 685-693, 1979.
- [6] S. C. Ma, P. Franco, and E. J. McCluskey, "An experimental chip to evaluate test techniques experiment results," in *Proceedings International Test Conference*, pp. 663-670, 1995.
- [7] M. Amyeen, S. Venkataraman, A. Ojha, and L. Sangbong, "Evaluation of the quality of N-detect scan ATPG patterns on a processor," in *Proc. IEEE International Test Conference*, pp. 669-678, 2004.
- [8] B. Koenemann, "LFSR-coded test patterns for scan design," in *Proc. the European Test Conference*, pp. 237-242, 1991.
- [9] S. Hellebrand, J. Rajski, S. Tarnick, S. Venkataraman, and B. Courtois, "Built-in test for circuits with scan based on reseeding of multiple-polynomial linear feedback shift registers," *IEEE Trans. Computers*, vol. 44, pp. 223-233, 1995.
- [10] C. Krishna and N. A. Toubia, "Reducing test data volume using LFSR reseeding with seed compression," in *Proc. International Test Conference*, pp. 321-330, 2002.
- [11] R. Tulloss, "Fault dictionary compression: Recognizing when a fault may be unambiguously represented with a single failure detection," in *Proc. International Test Conference*, pp. 368-370, 1980.
- [12] J. Richman and K. Bowden, "The modern fault dictionary," in *Proc. International Test Conference*, pp. 696-702, 1985.
- [13] M. Abramovici and M. A. Breuer, "Multiple fault diagnosis in combinational circuits based on an effect-cause analysis," *IEEE Transactions on Computers*, vol. C-29, pp. 451-460, 1980.

- [14] S. Venkataraman and W. Fuchs, "Deductive technique for diagnosis of bridging faults," in *Proc. International Conference on Computer-Aided Design*, pp. 562-567, 1997.
- [15] International Technology Roadmap for Semiconductors, <http://www.itrs.net/Links/2007ITRS/Home2007.htm>.
- [16] J. Rajski, J. Tyszer, M. Kassab, and N. Mukherjee, "Embedded deterministic test," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, pp. 776-792, May 2004.
- [17] B. Koenemann, "LFSR-coded test patterns for scan design," in *Proc. the European Test Conference*, pp. 237-242, 1991.
- [18] S. Hellebrand, J. Rajski, S. Tarnick, S. Venkataraman, and B. Courtois, "Built-in test for circuits with scan based on reseeding of multiple-polynomial linear feedback shift registers," *IEEE Trans. Computers*, vol. 44, pp. 223-233, 1995.
- [19] S. Hellebrand, H.-G. Liang, and H. J. Wunderlich, "A mixed mode BIST scheme based on reseeding of folding counters," in *Proc. International Test Conference*, pp. 778-784, 2000.
- [20] C. V. Krishna, A. Jas, and N. A. Touba, "Achieving high encoding efficiency with partial dynamic LFSR reseeding," *ACM Trans. on Design Automation of Electronic Systems*, vol. 9, pp. 500-516, Oct. 2004.
- [21] C. G. Cullen, *Linear Algebra with Applications*. Addison-Wesley, 1997.
- [22] C. V. Krishna and N. A. Touba, "Reducing test data volume using LFSR reseeding with seed compression," in *Proc. International Test Conference*, pp. 321-330, 2002.
- [23] Z. Wang and K. Chakrabarty, "Test-quality/cost optimization using output-deviation-based reordering of test patterns," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, pp. 352-365, 2008.
- [24] Z. Wang et al., "A seed-selection method to increase defect coverage for LFSR-reseeding-based test compression," in *Proc. ETS*, pp. 125-130, 2007.
- [25] K. H. Tsai et al., "STARBIST: Scan autocorrelated random pattern generation," in *Proc. Design Automation Conference*, pp. 472-477, 1997.
- [26] K. Hatayama et al., "Application of high-quality built-in test to industrial designs," in *Proc. ITC*, pp. 1003-1012, 2002.
- [27] N. Ahmed, M. Tehranipoor, and V. Jayaram, "Timing-based delay test for screening small delay defects," in *Proc. Design Automation Conference*, pp. 320-325, 2006.

- [28] X. Lin et al., "Timing-aware ATPG for high quality at-speed testing of small delay defects," in *Proc. Asian Test Symposium*, pp. 139-146, 2006.
- [29] P. C. Maxwell, I. Hartanto and L. Bentz, "Comparing functional and structural tests," in *Proc. International Test Conference*, pp. 400-407, 2000.
- [30] A. K. Vij, "Good scan= good quality level? well, it depends...," in *Proc. International Test Conference*, pp. 1195, 2002.
- [31] J. Gatej, L. Song, C. Pyron, R. Raina and T. Munns, "Evaluating ATE features in terms of test escape rates and other cost of test culprits," in *Proc. International Test Conference*, pp. 1040-1049, 2002.
- [32] T. Lv, H. Li, and X. Li, "Automatic selection of internal observation signals for design verification," in *Proc. VTS*, pp. 203-208, 2009.
- [33] J. Rearick and R. Rodgers, "Calibrating clock stretch during AC scan testing," in *Proc. International Test Conference*, pp. 266-273, 2005.
- [34] P. A. Thaker, V. D. Agrawal, and M. E. Zaghloul, "Register-transfer level fault modeling and test evaluation techniques for VLSI circuits," in *Proc. ITC*, pp. 940-949, 2000.
- [35] F. Corno, M. S. Reorda, and G. Squillero, "RT-level ITC'99 benchmarks and first ATPG result," *IEEE Design and Test of Computers*, vol. 17, pp. 44-53.
- [36] M. B. Santos, F. M. Goncalves, I. C. Teixeira, and J. P. Teixeira, "TL-based functional test generation for high defects coverage in digital systems," *Journal of Electronic Testing: Theory and Applications (JETTA)*, vol. 17, pp. 311-319.
- [37] W. Mao and R. K. Gulati, "Improving gate level fault coverage by RTL fault grading," in *Proc. ITC*, pp. 150-159, 1996.
- [38] S. Park, L. Chen, P. Parvathala, S. Patil and I. Pomeranz, "A functional coverage metric for estimating the gate-level fault coverage of functional tests," in *Proc. International Test Conference*, 2006.
- [39] I. Pomeranz, P. K. Parvathala, S. Patil, "Estimating the fault coverage of functional test sequences without fault simulation," in *Proc. Asian Test Symposium*, pp. 25-32, 2007.
- [40] T. E. Marchok, A. El-Maleh, W. Maly and J. Rajski, "A complexity analysis of sequential ATPG," *IEEE Transactions on Computer-Aided Design.*, vol. 15, pp. 1409-1423, 1996.
- [41] I. Pomeranz and S. M. Reddy, "LOCSTEP: A logic-simulation-based test generation procedure," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, pp. 544-554, 1997.

- [42] Biquad Infinite Impulse Response Filter, <http://open-cores.com/project,biquad/>.
- [43] B. J. Chalmers, *Understanding statistics*; CRC Press, 1987.
- [44] K. Y. Cho, S. Mitra and E. J. McCluskey, "Gate exhaustive testing," *in Proc. Int. Test Conf.*, pp. 771-777, 2005.
- [45] R. Guo et al., "Evaluation of test metrics: stuck-at, bridge coverage estimate and gate exhaustive," *in Proc. VLSI Test Symp.*, pp. 66-71, 2006.
- [46] S. Ravi and N. K. Jha, "Fast test generation for circuits with RTL and gate-level views," *in Proc. ITC*, pp. 1068-1077, 2001.
- [47] I. Ghosh and M. Fujita, "Automatic test pattern generation for functional RTL circuits using assignment decision diagrams," *in Proc. Design Automation Conference*, pp. 43-48, 1999.
- [48] H. Kim and J. P. Hayes, "High-coverage ATPG for datapath circuits with unimplemented blocks," *in Proc. ITC*, pp. 577-586, 1998.
- [49] O. Goloubeva, G. Jervan, Z. Peng, M. S. Reorda, and M. Violante, "High-level and hierarchical test sequence generation," *in Proc. HLDVT*, pp. 169-174, 2002.
- [50] N. Yogi and V. D. Agrawal, "Spectral RTL test generation for gate-level stuck-at faults," *in Proc. Asian Test Symposium*, pp. 83-88, 2006.
- [51] T. Hosokawa, R. Inoue, and H. Fujiwara, "Fault-dependent/independent test generation methods for state observable FSMs," *in Proc. Asian Test Symposium*, pp. 275-280, 2007.
- [52] R. Inoue, T. Hosokawa, and H. Fujiwara, "A test generation method for state-observable FSMs to increase defect coverage under the test length constraint," *in Proc. Asian Test Symposium*, pp. 27-34, 2008.
- [53] S. Dey and M. Potkonjak, "Non-scan design-for-testability of RT-level data paths," *in Proc. International Conference on Computer-Aided Design*, pp. 640-645, 1994.
- [54] R. B. Norwood and E. J. McCluskey, "Orthogonal scan: Low overhead scan for data paths," *in Proc. ITC*, pp. 659-668, 1996.
- [55] I. Ghosh, A. Raghunathan, and N. K. Jha, "Design for hierarchical testability of RTL circuits obtained by behavioral synthesis," *in Proc. IEEE International Conference on Computer Design*, pp. 173-179, 1995.

- [56] I. Ghosh, A. Raghunathan, and N. K. Jha, "A design for testability technique for RTL circuits using control/dataflow extraction," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, pp. 706-723, 1998.
- [57] H. Wada, T. Masuzawa, K. K. Saluja, and H. Fujiwara, "Design for strong testability of RTL data paths to provide complete fault efficiency," in *Proc. IEEE International Conference on VLSI Design*, pp. 300-305, 2000.
- [58] H. Fujiwara, H. Iwata, T. Yoneda, and C. Y. Ooi, "A nonscan design-for-testability method for register-transfer-level circuits to guarantee linear- depth time expansion models," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, pp. 1535-1544, Nov. 2008.
- [59] J. P. Grossman, J. K. Salmon, C. R. Ho, D. J. Ierardi, B. Towles, B. Batson, J. Spengler, S. C. Wang, R. Mueller, M. Theobald, C. Young, J. Gagliardo, M. M. Deneroff, R. O. Dror, and D. E. Shaw, "Hierarchical simulation-based verification of Anton, a special-purpose parallel machine," in *Proc. IEEE International Conference on Computer Design*, pp. 340-347, 2008.
- [60] D. A. Mathaikutty, S. Ahuja, A. Dingankar, and S. Shukla, "Model-driven test generation for system level validation," in *Proc. HLDVT*, pp. 83-90, 2007.
- [61] O. Guzey and L.-C. Wang, "Coverage-directed test generation through automatic constraint extraction," in *Proc. HLDVT*, pp. 151-158, 2007.
- [62] H. Fang, K. Chakrabarty, A. Jas, S. Patil, and C. Trimurti, "RT-level deviation-based grading of functional test sequences," in *Proc. VTS*, pp. 264-269, 2009.
- [63] Z. Wang, H. Fang, K. Chakrabarty, and M. Bienek, "Deviation-based LFSR reseeding for test-data compression," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, pp. 259-271, 2009.
- [64] Z. Wang and K. Chakrabarty, "An efficient test pattern selection method for improving defect coverage with reduced test data volume and test application time", *Asian Test Symposium*, pp. 333-338, 2006.
- [65] K. Y. Cho, S. Mitra, and E. J. McCluskey, "Gate exhaustive testing," in *Proc. ITC*, pp. 771-777, 2005.
- [66] Y. Huang, "On n-detect pattern set optimization," in *Proc. International Symposium on Quality of Electronic Design*, pp. 445-450, 2006.
- [67] H. Fang, K. Chakrabarty, and R. Parekhji, "Bit-operation-based seed augmentation for LFSR reseeding with high defect coverage," *accepted for publication in IEEE Asian Test Symposium*, 2009.
- [68] [Online]. Available: <http://www.iwls.org/iwls2005/benchmarks.html>.

- [69] M. Yilmaz et al., “Test-pattern grading and pattern selection for small-delay defects,” in *Proc. VTS*, pp. 233-239, 2008.
- [70] H. Tang et al., “Defect aware test patterns,” in *Proc. DATE*, pp. 450-455, 2005.
- [71] Z. Navabi, *VHDL: Analysis and Modeling of Digital Systems*, New York; McGraw-Hill Companies, 1997.
- [72] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel, “Characterizing the effect of transient faults on a high-performance processor pipeline,” in *Proc. International Conference on Dependable Systems and Networks*, pp. 61-70, 2004.
- [73] N. J. Wang and S. J. Patel, “Restore: symptom based soft error detection in microprocessors,” in *Proc. International Conference on Dependable Systems and Networks*, pp. 30-39, 2005.
- [74] M. Maniatakos, N. Karimi, Y. Makris, A. Jas, and C. Tirumurti, “Design and evaluation of a timestamp-based concurrent error detection method (CED) in a modern microprocessor controller,” in *Proc. International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2008.
- [75] N. Karimi, M. Maniatakos, A. Jas, and Y. Makris, “On the correlation between controller faults and instruction-level errors in modern microprocessors,” in *Proc. International Test Conference*, 2008.
- [76] Documentation for the Parwan processor, <http://mesdat.ucsd.edu/~lichen/260c/parwan/>.
- [77] B. Benware, C. Schuermyer, S. Ranganathan, R. Madge, P. Krishnamurthy, N. Tamarapalli, K-H Tsai and J. Rajski, “Impact of multiple-detect test patterns on product quality,” in *Proc. International Test Conference*, pp. 1031-1040, 2003.
- [78] H. Tang, G. Chen, S. Reddy, C. Wang, J. Rajski, and I. Pomeranz, “Defect aware test patterns,” in *Proc. Design, Automation, and Test in Europe*, pp. 450-455, 2005.
- [79] N. Yazdani and Z. Ozsoyoglu, “Sequence matching of images,” in *Proc. International Conference on Scientific and Statistical Database Systems*, pp. 53-62, 1996.
- [80] M. Lu and H. Lin, “Parallel algorithms for the longest common subsequence problem,” in *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, pp. 835-848, 1994.
- [81] G. Kalton, “Introduction to survey sampling,” *SAGE University Paper series on Quantitative Applications in the Social Sciences*, series no. 07-035, Beverly Hills; SAGE Publications, 1983.

- [82] R. Guo, S. Mitra, E. Amyeen, J. Lee, S. Sivaraj, and S. Venkataraman, "Evaluation of test metrics: Stuck-at, bridge coverage estimate and gate exhaustive," in *Proc. VTS*, pp. 66-71, 2006.
- [83] J. Gatej et al., "Evaluating ATE features in terms of test escape rates and other cost of test culprits," in *Proc. ITC*, pp. 1040-1049, 2002.
- [84] T. Vo et al., "Design for board and system level structural test and diagnosis," in *Proc. ITC*, pp. 409-418, 2006.
- [85] L. Wand et al., "A self-test and self-diagnosis architecture for boards using boundary scans," in *ETC*, pp. 119-126, 1989.
- [86] D. Manley and B. Eklow, "A model based automated debug process," in *IEEE Board Test Workshop*, pp. 1-7, 2002.
- [87] M. Abramovici et al., "A reconfigurable design-for-debug infrastructure for SoCs," in *Proc. Design Automation Conference*, pp. 7-12, 2006.
- [88] J.-S. Yang and N. A. Touba, "Expanding trace buffer observation window for in-system silicon debug through selective capture," in *Proc. VTS*, pp. 345-351, 2008.
- [89] A. Carbine and D. Feltham, "Pentium®Pro processor design for test and debug," in *Proc. ITC*, pp. 294-303, 1997.
- [90] H. Hao and K. Bhabuthmal, "Clock controller design in *SuperSPARC™* II microprocessor," in *Proc. IEEE International Conference on Computer Design*, pp. 124-129, 1995.
- [91] C. OFarrill et al., "Optimized reasoning based diagnosis for non-random, board-level, production defects," in *Proc. ITC*, pp. 173-179, 2005.
- [92] Z. Zhang et al., "Board-level fault diagnosis using Bayesian inference," in *Proc. VTS*, pp. 244-249, 2010.
- [93] G. Shafer, *A Mathematical Theory of Evidence*, Princeton University Press, Princeton, New Jersey, 1976.
- [94] FICO. Xpress-MP., <http://www.fico.com/en/Products/DMTools/Pages/FICO-Xpress-Optimization-Suite.aspx>.
- [95] I. Majzik, "Software monitoring and debugging using compressed signature sequences," in *Proc. of the 22nd EUROMICRO Conf.*, pp. 311-318, 1996.
- [96] H. Hao and R. Avra, "Structured design-for-debug - the *SuperSPARC™* II methodology and implementation," in *Proc. ITC*, pp. 175-183, 1995.

- [97] I. Parulkar et al., "DFX of a 3rd generation, 16-core/32-thread *UltraSPARCTM* CMT microprocessor," in *Proc. ITC*, 2008.
- [98] Y-C. Lin et al., "Pseudo-functional testing," *IEEE Trans. CAD*, vol. 25, pp. 1535-1546, 2006.
- [99] Q. Wu and M. S. Hsiao, "Efficient ATPG for design validation based on partitioned state exploration histories," in *Proc. VTS*, pp. 389-394, 2004.
- [100] Q. Wu and M. S. Hsiao, "State variable extraction and partitioning to reduce problem complexity for ATPG and design validation," *IEEE Trans. CAD*, vol. 25, pp. 2275-2282, 2006.
- [101] C. Werner et al., "Crosstalk noise in future digital CMOS circuits," in *Proc. DATE*, pp. 331-335, 2001.
- [102] D. B. West, *Introduction to graph theory - second edition*. Prentice Hall, 2001.
- [103] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, pp. 146-160, 1972.
- [104] Z. Zhang, Z. Wang, X. Gu, and K. Chakrabarty, "Physical defect modeling for fault insertion in system reliability test," in *Proc. ITC*, pp. 1-10, 2009.

Biography

Hongxia Fang

hf12@duke.edu

PERSONAL DATA

Date of birth: August 3, 1980.

Place of birth: Tianmen, Hubei, China.

EDUCATION

Doctor of Philosophy, Duke University, USA, expected 2011.

Master of Science, Institute of Computing Technology, Chinese Academy of Sciences, China, 2005.

Bachelor of Engineering, Nankai University, China, 2002.

PATENT

Z. Wang, X. Gu, Z. Wang, H. Fang, “System and Method for Executing Functional Scanning in an Integrated Circuit Environment”, *filed with the United States Patent Office*, August 2010.

PUBLICATIONS

• Journal Articles

1. Z. Wang, H. Fang, K. Chakrabarty and M. Bienek, “Deviation-based LFSR reseeding for test-data compression”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, February 2009, pp. 259–271.
2. H. Fang, K. Chakrabarty and H. Fujiwara, “RTL DFT techniques to enhance defect coverage for functional test sequences”, *Journal of Electronic Testing: Theory and Applications (JETTA)*, vol. 26, April 2010, pp. 151–164.

- **Refereed Conference Papers**

1. H. Fang, K. Chakrabarty, A. Jas, S. Patil and C. Tirumurti, “RT-Level Deviation-Based Grading of Functional Test Sequences”, *Proc. IEEE VLSI Test Symposium*, pp. 264-269, 2009.
2. H. Fang, K. Chakrabarty and R. Parekhji, “Bit-operation-based seed augmentation for LFSR reseeding with high defect coverage”, *Proc. IEEE Asian Test Symposium*, pp. 331-335, 2009.
3. H. Fang, K. Chakrabarty and H. Fujiwara, “RTL DFT techniques to enhance defect coverage for functional test sequences”, *Proc. IEEE International High Level Design Validation and Test Workshop*, pp. 160-165, 2009. (Invited paper)
4. H. Fang, Z. Wang, X. Gu and K. Chakrabarty, “Mimicking of functional state space with structural tests for the diagnosis of board-level functional failures”, *Proc. IEEE Asian Test Symposium*, pp. 421-428, 2010.
5. H. Fang, Z. Wang, X. Gu and K. Chakrabarty, “Deterministic test for the reproduction and detection of board-level functional failures”, *Proc. IEEE/ACM Asia South Pacific Design Automation Conference*, pp. 491-496, 2011.
6. H. Fang, Z. Wang, X. Gu and K. Chakrabarty, “Ranking of Suspect Faulty Blocks using Dataflow Analysis and Dempster-Shafer Theory for the Diagnosis of Board-Level Functional Failures”, *Proc. IEEE European Test Symposium*, 2011.

- **Submitted Journal Articles**

1. H. Fang, K. Chakrabarty, A. Jas, S. Patil, and C. Tirumurti, “Functional Test Sequence Grading at Register-Transfer Level”, submitted to *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2011.
2. H. Fang, K. Chakrabarty, Z. Wang and X. Gu, “Reproduction and detection of board-level functional failures”, submitted to *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2011.