Exploring Temporal and Spatial Correlations on Circuit Variables for Enhancing Simulation-based Test Generation

Xiaoding Chen

Dissertation submitted to the Faculty of

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Engineering

Dr. Michael S. Hsiao, Chair/Advisor

Dr. Stephen Edwards

Dr. Guoquan Lu

Dr. Tom L. Martin

Dr. Sandeep K. Shukla

26 May, 2006

Bradley Department of Electrical and Computer Engineering

Virginia Polytechnic Institute and State University

Blacksburg, Virginia 24061

Keywords: Simulation, ATPG, Correlation, State Variables, BIST

Copyright © 2006, Xiaoding Chen

Exploring Temporal and Spatial Correlations on Circuit Variables for Enhancing Simulation-based Test Generation

Xiaoding Chen

Abstract

The ever-increasing complexity and size of current circuit designs have made testing and verification major bottlenecks in the design flow of VLSI (Very Large Scale Integrated) circuits. Statistics show that more than 70% of the design effort can be spent on functional verification and manufacturing testing. This percentage is expected to increase in the future if no significant strides in these areas are made. In this dissertation, we target three related problems in simulation-based Design Verification and Testing: Sequential ATPG (Automatic Test Pattern Generation), Unbounded Model Checking (UMC) of safety properties, and low power testing for full-scan sequential circuits. We model these three problems as simulation-based pattern generation problems and exploit novel ATPG algorithms to increase the effectiveness of sequential ATPGs.

The main challenge for fault/error detection in sequential circuits is the large number of flip-flops (FFs) in modern designs. Due to the large number and variable length of test sequences required for such circuits, the existing deterministic ATPG algorithms fail to achieve high test coverages. Such algorithms typically work by first unrolling the sequential circuit and then performing frequent backtracking to generate test vectors for fault detection. For the hard-to-detect faults, these schemes either run out of memory or require a huge computational effort. We show that simulation-based ATPGs, on the other hand, scale very well for large circuits as they perform only forward simulation. A fundamental problem associated with simulation-based ATPGs is to avoid exhaustive circuit simulation, which is impractical for large designs in the real world, by choosing high quality test vectors that achieve a high test coverage within a low simulation time. We tackle this

primary problem by exploiting different correlation-based heuristics.

The intuition behind using correlation-based heuristics is to better guide the pattern generation engine such that the specific objective of either fault detection or property verification in UMC or minimizing power consumption during the testing, is achieved in an efficient manner without resorting to exhaustive simulation. In particular, we model and explore the following correlations: (1) temporal correlations, i.e. correlations on each primary input (PI) in different time frames, and (2) spatial correlations, i.e. correlations among different FFs in the same time frame. We employ temporal correlations in the context of pattern generation of a built-in-self-test (BIST) architecture and we explore spatial correlations to guide a logic-simulation-based sequential ATPG and low power scan test generation. Experimental results on ISCAS and ITC benchmark circuits have shown that those correlations can enhance the simulation to discover more faults or design errors in a significantly shorter time.

Acknowledgements

It is a pleasure to acknowledge all the people who made this work possible. I would like to express my sincere thanks to my advisor Dr. Michael S. Hsiao, for his inspiration and support throughout my graduate program. I would like to thank Dr. Edwards, Dr. Lu, Dr. Martin, and Dr. Shukla for serving on my thesis committee. Last but not the least, I thank my friends and relatives for their emotional support and encouragement.

Xiaoding Chen
May 2006

Contents

Contents v

List of Figures ix

List of Tables xii

1 Introduction 1

1.1 Introduction to Testing . . . 1

1.1.1 Single Stuck-at Fault Model . . . 2

1.2 Sequential ATPG . . . 3

1.2.1 Deterministic ATPGs . . . 4

1.2.2 Simulation-based ATPGs . . . 6

1.3 Applications of Simulation-based ATPGs . . . 6

1.4 Built-In-Self-Test . . . 7

1.5 Contribution and Outline of the Dissertation . . . 8

2 Background 11

2.1 Logic-Simulation-Based Sequential ATPG . . . 11

2.1.1 NL-ATPG vs. PL-ATPG . . . 12

2.2 Logic BIST . . . 14

2.2.1 Scan-based Design . . . 15

2.2.2 Low Power Schemes for Scan-based Logic BIST . . . 16

2.3 Correlations . . . 18

2.3.1 Definition of Correlations . . . 19

2.3.2 Examples of Correlations . . . 19

2.3.3 How to Learn Correlations . . . 20

2.3.4 How to Use Correlations . . . 21

3 Characteristic Faults and Spectral Information for Logic BIST 23

3.1 Introduction . . . 24

3.2 Overview and Motivation . . . 25

3.3 Computation of Characteristic Faults . . . 26

3.4 State Relaxation . . . 29

3.4.1 Review of Support Sets . . . 29

3.4.2 Relaxation in Sequential Circuit . . . 30

3.5 Experimental Results . . . 34

3.6 Summary . . . 38

4 Testing Embedded Sequential Cores in Parallel Using Spectrum-Based BIST 39

4.1 Introduction . . . 40

4.2 Overview and Motivation . . . 41

4.3 Interleaved State Relaxation & DFT Insertion . . . 44

4.3.1 BIST Architecture . . . 44

4.3.2 Additional Terminologies . . . 47
4.3.3 Algorithm for Interleaved State Relaxation . . . 49
4.3.4 Delay Network . . . 53
4.3.5 Switching Network . . . 54
4.3.6 Algorithm for Spectral Compression . . . 55
4.4 Experimental Results . . . 57
4.4.1 Experimental Results on ISCAS89 & ITC99 Benchmark Circuits . . . 57
4.4.2 Quantitative Analysis on Speedup & Storage Reduction . . . 61
4.5 Summary . . . 64

5 Characteristic States and Cooperative Game Based Search 66
5.1 Introduction . . . 67
5.2 Background and Motivations . . . 68
5.2.1 Motivation Behind Our Approach . . . 70
5.2.2 Overview of Game Theory . . . 74
5.2.3 Basic Overview of Genetic Algorithms . . . 77
5.3 The Proposed Approach . . . 78
5.3.1 Cooperative Search Using Game Theory . . . 78
5.3.2 Characteristic States . . . 81
5.4 Experimental Results . . . 84
5.5 Summary . . . 88

6 A Hybrid Property Checking Engine 90
6.1 Introduction . . . 91

6.2 Background and Motivations . . . 93
6.3 Our Framework . . . 94
6.3.1 Fitness Function for GA . . . 95
6.4 Our Approach to Verification . . . 96
6.4.1 Approximate Preimage Directed Logic Partitioning . . . 97
6.4.2 LH & GSO Based Feedback Mechanism . . . 101
6.5 Experimental Results . . . 106
6.6 Summary . . . 108

7 Energy-Efficient Logic BIST Based on State Correlation Analysis 109

7.1 Introduction . . . 110
7.2 Preliminaries . . . 112
7.2.1 Structure of LT-RTPG . . . 113
7.3 Computing State Spatial Correlation Matrix . . . 114
7.3.1 Underlying Methodology . . . 115
7.3.2 Algorithms . . . 118
7.4 Experimental Results . . . 122
7.5 Summary . . . 126

8 Conclusion 127

Bibliography 129

List of Figures

1.1 Single Stuck-at Fault Model 3

1.2 Sequential Circuit and Its Combinational Logic 4

1.3 Unrolling Sequential Circuit for 3 Time Frames 5

1.4 Comparison between Conventional Test and BIST 7

2.1 State Partition Examples 13

2.2 Scan Cell and Serial Scan Architecture 16

2.3 Pattern Application on Scan-based Design 17

2.4 Correlations: Example I 20

2.5 Correlations: Example II 21

2.6 Employing Correlations into Test Generation 22

3.1 Overall Framework 27

3.2 Algorithm for Extracting Characteristic Faults 28

3.3 Algorithm for Computing Minimal Support Set 31

3.4 Good/Faulty Circuit Values at Time Frame 5 32

3.5 Good/Faulty Circuit Values at Time Frame 4 33

4.1 Interleaved State Relaxation 43

4.2 BIST Architecture 44
4.3 Using Delay Unit and Switching to Reduce Conflicts 45
4.4 Delay Network and Internal Connections 46
4.5 A Merging Example with Delay and Switching Network 47
4.6 Algorithm for Interleaved State Relaxation 51
4.7 Algorithm for Performing Spectral Compression 56

5.1 Different Types of Correlations 70
5.2 Comparison of the Framework Between Previous Work and Current Work 71
5.3 Illustration of Cooperative Search 72
5.4 Illustration of Closed Loop Genetic Algorithm 73
5.5 State Space Search via a Two-player Game 79
5.6 Algorithm for State Condensation 83

6.1 Framework of the Proposed Hybrid Model Checker 95
6.2 Algorithm for BDD Approximation 99
6.3 Algorithm for Extracting Logic Correlations 100
6.4 An Example on the Construction of Logic Matrix 102
6.5 Algorithm for Genetic Spot Optimization 105

7.1 Architecture of LT-RTPG 113
7.2 Relaxed Excitation Seqs for Undetected Faults f_1, f_2 116
7.3 Comparison of Scan Chain before and after Altering 116
7.4 Compatibilities before and after Scan Chain Altering 117
7.5 Toggle Probability before and after Scan Chain Altering 118

7.6 Algorithm for Configuring the Scan Chain 119

7.7 Algorithm for Constructing Spatial Correlation Matrix 120

7.8 Algorithm for Selecting Scan Flip-flops 121

7.9 Algorithm for Scan Chain Reordering 122

7.10 Algorithm for Correction 123

List of Tables

3.1 Comparison of Fault Coverages 35

3.2 Fault Coverage for Varying Number of Characteristic Faults 37

3.3 Computational Cost and Storage Requirement 37

4.1 Applying a Single Test Pattern for Cluster *i* 47

4.2 Different Modes for the Chip and a Cluster 47

4.3 Conflicts after Merging for Different Direct Input Mappings 53

4.4 Fault Detection of Different Techniques 57

4.5 Storage in Bit for Testing a Single Core by Spectral BIST 58

4.6 Comparison of Fault Detection, Test Application Time and Storage Requirement By Testing Cluster as a Whole 59

5.1 Payoff Table for the Prisoner's Dilemma 76

5.2 Comparison of Coverage and Coverage Deviation with/without Condensation & with One/Two Players 85

5.3 Comparison of Fault Coverage on Large Benchmark Circuits 87

5.4 Comparison of State Coverage on Each Partition Under Different Traversal Methods for b13 87

5.5 Coverage of Gate Substitution Error 89

6.1 Verification Results with/without GSO and/or LH Injection 107

6.2 Verification Results Compared with Other Model Checkers 107

7.1 Toggle Probability 114

7.2 Comparison of Fault Coverages, Heat Dissipation and Energy Efficiency 124

Chapter 1

Introduction

The growing advances in modern VLSI technology have led to an increased complexity in today's hardware systems. The complexity of large designs pose serious challenges to pre-silicon verification and post-silicon testing. The discrepancies in a manufactured chip may be due to the design bugs, manufacturing defects or the manner in which test is applied. In order to address these problems, it is necessary to verify the design and generate test cases that can detect the design errors as well as manufacturing defects without damaging the chip during the test.

1.1 Introduction to Testing

Testing of a digital system is an experiment in which the digital system is exercised and its resulting response is compared with the expected response to ascertain whether it behaved correctly. Generally, a set of pre-designed stimuli, *test pattern set*, and the corresponding response, *test response*, define a test experiment for the system under test. In this dissertation, *testing* refers to manufacturing test, i.e. test targeting manufacturing defects. On the other hand, a functional test aimed to catch design errors is referred to as *verification*.

Test evaluation is usually done in the context of a *fault model*. *Fault simulation* is carried out to compute the response of the circuit in the presence of faults. A fault is detected when the response of the faulty-circuit differs from the expected response of the fault-free circuit. An important metric is *fault coverage*, which evaluates the effectiveness, or the quality, of a test. It is defined as the ratio between the number of faults detected and the total number of faults in the circuit-under-test (CUT).

1.1.1 Single Stuck-at Fault Model

Logical faults represent the effect of physical faults on the behavior of the modeled system. Unless explicitly stated, most of the literature in the testing area assumes that at most one logical fault is present in the system. This single-fault assumption simplifies the test generation problem. Practically, even when multiple faults are present, the test derived under the single-fault assumption is usually applicable for the detection of multiple faults by composing the individual tests designed for each single fault.

Some well-known fault models are *single stuck-at fault* (SSF), *bridging fault*, and *delay fault*. In this dissertation, ATPG algorithms are developed targeting the *single stuck-at fault* model, the simplest but most widely used fault model. The effect of the single stuck-at fault is as if the faulty node is tied to either *VCC* (s-a-1), or *GND* (s-a-0), shown in Figure 1.1. The advantages of using single stuck-at fault model include:

1. It represents many different physical faults;
2. It is independent of technology;
3. Experiments have shown that tests that detect SSFs detect many other faults as well;
4. The total number of SSFs in the circuit is linear to the size of the circuit;
5. Many other fault models can be represented as a combination of SSFs.

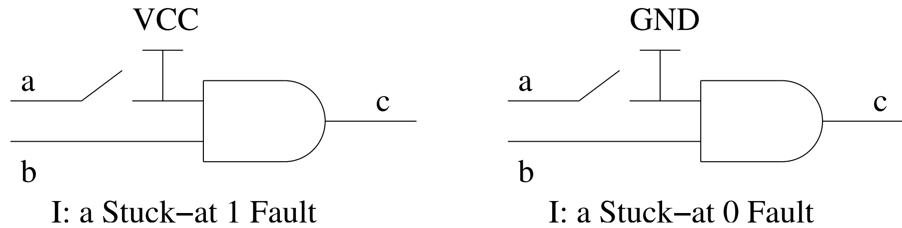


Figure 1.1. Single Stuck-at Fault Model

1.2 Sequential ATPG

ATPG is a design automation tool that attempts to find an input (or test) sequence that, when applied to a digital circuit, distinguishes between the correct circuit behavior and the faulty circuit behavior caused by a particular fault. The effectiveness of ATPG is measured by the fault coverage achieved for the fault model.

ATPG for combinational circuits targeting SSFs has been a well-studied problem. Commercial combinational ATPG tools, such as TetraMAX [1] from Synopsys, achieve an average of 99% test efficiency on real-world industrial designs. Unlike combinational circuits, where no memory elements (flip-flops, latches, etc.) exist, and the detection of a target fault can be achieved by a single vector, sequential circuits contain a portion of memory elements, and the detection of a target fault requires an ordered vector sequence. The lengths of the vector sequences for detecting different target faults in a sequential circuit may vary.

ATPG for sequential circuits remains an open and difficult problem. For sequential circuits without a global reset state, or with a very biased number of PIs and FFs (# PIs<<# FFs), test coverages are generally very low, sometimes below 30%. It is still not clear to researchers whether there exists a test sequence that can distinguish between the behaviors of the good circuit and the faulty circuit if the undetected faults are present. In other words, it is possible that some faults in the undetected fault set are untestable faults whose presence does not change the input/output behavior

of the circuit. Research in sequential ATPG has been carried out in two directions: (1) targeting at increasing the test coverage; or (2) targeting at proving more untestable faults. The existing sequential ATPG tools can be classified into two categories: (1) deterministic ATPGs; and (2) simulation-based ATPGs.

1.2.1 Deterministic ATPGs

Deterministic sequential ATPGs have been devised on the basis of fundamental combinational ATPG algorithms. The Iterative Logic Array (ILA) model of the sequential CUT is constructed by unrolling the circuit for a certain number of time frames. An example of unrolling the sequential circuit into the ILA model by replicating the combinational circuitry for multiple time frames is shown in Figure 1.2 and Figure 1.3. ILA models a combinational circuit that mimics the functionality of the sequential circuit for a certain number of time frames.

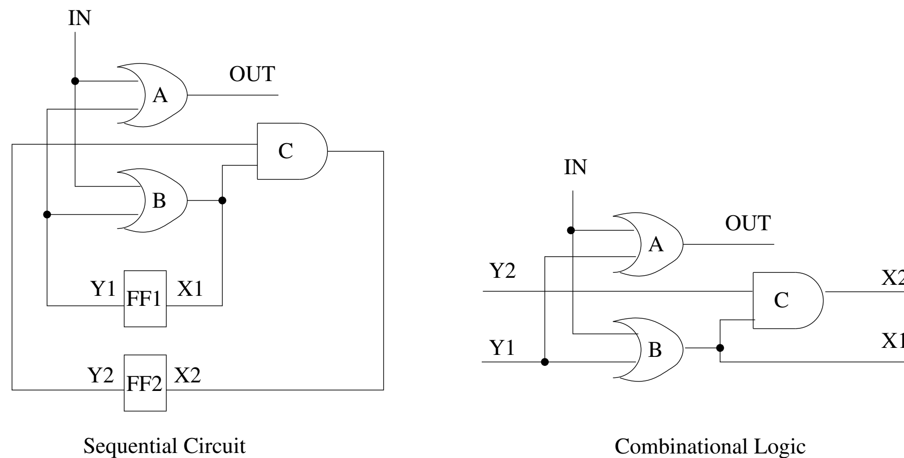


Figure 1.2. Sequential Circuit and Its Combinational Logic

A deterministic ATPG starts with a copy of the combinational circuit and sets it to time-frame 0. The target fault is injected in time-frame 0 and a basic combinational ATPG algorithm (such as the D Algorithm [2]) is invoked to justify the excitation of the fault on the present state variables

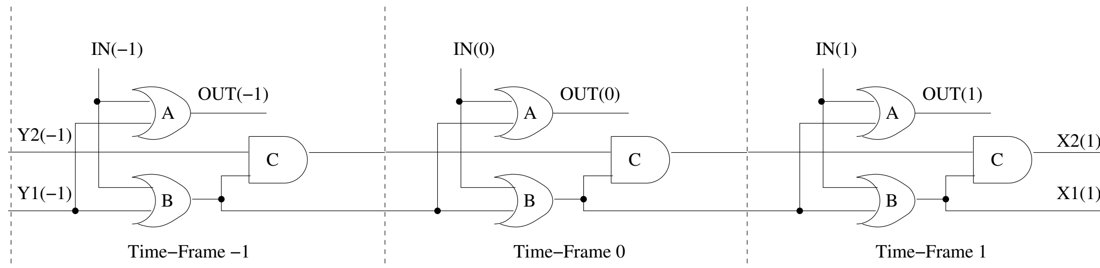


Figure 1.3. Unrolling Sequential Circuit for 3 Time Frames

and propagate the fault effect to the next state variables. If there are any values required on the present state variables, a new copy of the combinational logic is appended before the present state variables in the previous time frames; if the fault effects are not present on Primary Outputs (POs), but on the next state variables, a new copy of the combinational logic is appended after the next state variables in the future time frames. The expansion of time-frames continues until there is no present state variables to be justified backwards, and the fault effect has appeared on a PO.

Backtracking is the main bottleneck in combinational ATPG algorithms. When justification is performed backwards, some signals may appear to have conflicting logic values, which means some decisions made previously during the search are wrong. This is where backtracking occurs. Backtracking is performed by flipping the value of one of the previous decisions. Backtracking is very costly in combinational ATPG algorithms, and when dealing with ILA for deterministic sequential ATPGs, it can be even more costly because the ILA contains several copies of the circuit with a much larger number of variables. Most deterministic sequential ATPG tools cannot achieve very high test coverages because they frequently abort on hard faults when the number of backtracks exceeds a predefined limit.

1.2.2 Simulation-based ATPGs

Simulation-based sequential ATPG approaches are built upon either a logic simulator or fault simulator. These approaches combine simulation with the guidance of objective functions/fitness functions. The first few vector sequences, or tests, are usually randomly generated. Based on the previously generated tests, a new test is generated. Logic/fault simulation is then performed on the new test. An objective/fitness function is associated with the simulation results of the new test to evaluate whether the test can bring any value to fault detection (by lowering the cost of fault detection or by increasing the fitness of fault detection). If so, the test will be included in the final test set; otherwise, it will be discarded. The process repeats until the objective is achieved or the size of the final test set reaches a predefined number.

Only forward simulation is performed in simulation-based ATPGs, with no backtraces or backtracks involved. Thus, simulation-based ATPGs cannot identify untestable faults, but it is scalable and simple. Simulation-based sequential ATPG tools are able to handle large sequential circuits, in which deterministic ATPG tools can potentially fail. Since the input space is huge, it is not applicable for performing exhaustive simulation. The efficiency of simulation-based ATPGs largely depends on the quality of the generated vectors. The objective/fitness function plays a key role in guiding the simulation to pick useful vectors.

1.3 Applications of Simulation-based ATPGs

Fault-simulation-based ATPGs are aimed at generating manufacturing tests for SSFs. Logic-simulation-based ATPGs, where no fault model is involved, are applicable to manufacturing test, design validation and model checking. The key is to develop some metrics as the cost/fitness function to guide the search, which are closely related to the objective (fault detection, property verification or power minimization).

1.4 Built-In-Self-Test

Built-In-Self-Test (BIST) [3] is defined as "a design-for-test technique in which testing is accomplished through built-in hardware features". The methodology of BIST is shown in the right half of Figure 1.4: a BIST controller generates the test patterns, controls the test clock on the CUT, and collects and explains the test responses. This makes the external test interface much more compact and simpler than the conventional test approach: the number of pins that exchange data with the tester is greatly reduced and the test clock is generated on chip. Through BIST, designers start to regard the test as part of the system function. An optimistic estimation [4] for the hardware overhead of BIST is around 3%-5%.

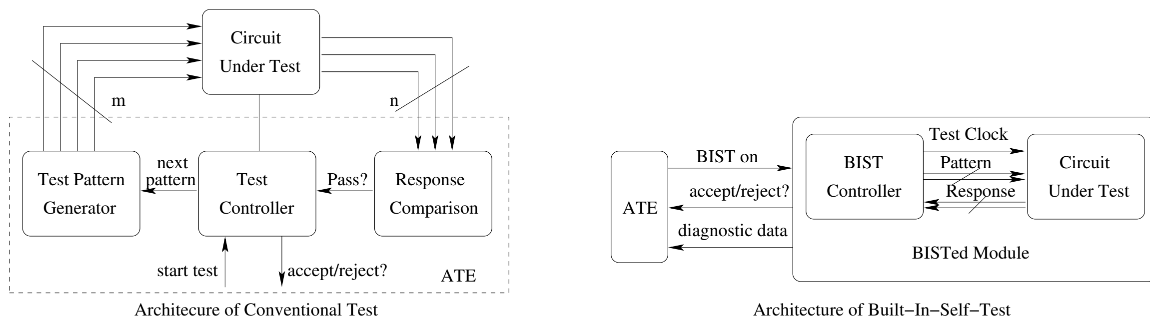


Figure 1.4. Comparison between Conventional Test and BIST

BIST alleviates the burden on purchasing expensive Automatic-Test-Equipments (ATEs), whose prices are proportional to the number of pins on the tester. BIST is also a good alternative to perform at-speed [5] testing when the technology (clock rate, propagation delays, etc.) of ATEs lags behind that of CUT. Furthermore, BIST is a promising solution for testing embedded cores in the SOC environment, whose testabilities are greatly reduced due to limited accessibility. Logic BIST with a good algorithm for pattern generation makes the testing of sequential cores in the SOC easier.

1.5 Contribution and Outline of the Dissertation

In the next chapter, we provide some background on the recent techniques that have been investigated for simulation-based ATPGs. We start with some basic explanations on ATPGs and Built-In-Self-Test. Then we compare the difference between non-partitioned logic-simulation-based ATPGs and partitioned logic-simulation-based ATPGs. In the latter part of the chapter, we introduce some low power BIST schemes. Finally, we explain how correlation works as a good heuristic in guiding the simulation-based pattern generation.

Our first contribution is the use of spectral analysis [6] to extract the temporal correlations for each primary input along different time frames, presented in Chapter 3. Analogous to signal processing, the test vector set is transformed from the temporal domain to the frequency domain via Hadamard transformation. "Noise" embedded in the spectrum is filtered and the "cleaner" spectrum is transformed back to the temporal domain for more efficient test generation. Characteristic faults and state relaxation are the two approaches employed to filter "noise" on the temporal domain. This ATPG engine fits well for testing the embedded sequential cores in the system-on-a-chip (SOC) environment, where an embedded controller or processor makes it possible for the processing unit to preform the correlation analysis and generate patterns to test the rest of the chip. The novel spectral analysis based BIST architecture in Chapter 3 is also extended in Chapter 4 such that sequential cores on the SOC can be tested in parallel [7] to reduce the test time, which is a major factor affecting the test cost. Experimental results show that the same level of fault coverage can be achieved for each individual core with negligible hardware overhead, while the test application time can be reduced by up to 4 times, and the test data storage requirement is reduced by up to 42%.

Secondly, a novel logic-simulation-based ATPG engine [8] is proposed to explore the partitioned state space in Chapter 5. Two types of spatial correlations among state variables are studied: (1) spatial correlations among state variables when they are acting as Pseudo Primary Inputs (PPIs)

to control other variables' values; and (2) spatial correlations among state variables when they are acting as Pseudo Primary Outputs (PPOs) to be controlled by others. Two different partition sets on the state variables are obtained based on the two different types of spatial correlations. Representing each partition set as a player, a two-player model based on Game Theory is employed to investigate how the interactions between the two players can help the search to detect more faults.

Our third contribution is the application of the logic-simulation-based ATPG mentioned above to design verification in Chapter 6. Verifying safety properties can also be viewed as a state justification problem. Facing the "state explosion" problem, formal methods are not applicable to unbounded model checking. On the other hand, simulation is the main scalable workhorse for verification today. A new hybrid property checking engine, which combines formal-method-based partitioning with logic-simulation is proposed to traverse the partitioned state space, with the hope that simulation can hit the target state. The hybrid verifier employs a property-dependent formal partitioning approach via a fast preimage computation. The motivation for such a partitioning is to weed out "noise" embedded in the huge search space. In particular, correlations among state variables in the same partition will be favored by the search; on the other hand, correlations among state variables in different partitions are considered as "noise" and will be ignored. An intelligent partitioning can uncover the correlations which are critical for verifying the target property, by putting closely related state variables into the same partition. The new approximate preimage directed logic partitioning aims at extracting the dominant correlations among the state variables in supporting a given property to better guide the search. Experimental results on large benchmark circuits have shown that the proposed hybrid verifier achieves equal or better results compared with the existing formal verification and simulation-based techniques.

Furthermore, in Chapter 7, we present a new low power BIST for sequential circuits [9]. We consider spatial correlations among FFs for low power test generation on full-scan sequential circuits. Our work addresses a solution for the scan architecture, which can reduce the test power with-

out increasing the test time or sacrificing the test coverage. Targeting dynamic power, we count the number of switches in the circuit-under-test as a power measurement. The majority of power consumption occurs during the shifting phase of the test application, where the "rippling effect" frequently happens, though most of the switches during the shifting phase do not contribute to fault detection. The relative positions of FFs along the scan chain determine the spatial correlations among FFs from the ATPG's point of view; spatial correlations among FFs from the CUT's point of view are embedded in the test vector set where fault detections are observed. To make the test pattern low power, we try to match the two spatial correlation profiles via scan chain reordering, such that switches that do not contribute to fault detection are suppressed. Our experiments show that the same or higher fault coverage can be achieved with less energy (and average power) - average power is reduced by 48.5%, with the maximum reduction of 73%.

Finally, we conclude the dissertation in Chapter 8.

Chapter 2

Background

In this chapter, we provide the necessary background in sequential ATPG and logic BIST to demonstrate our contribution in the area. In the first part of the chapter, we explain the different techniques to perform simulation-based sequential ATPG. In particular, we explain the basics of partitioned state traversal with its application on pattern generation. In the latter part of the chapter, we point out the high power dissipation problem faced by scan-based logic BIST and discuss some low power BIST schemes proposed in the literature. Finally, we explain how correlation works as a good heuristic in guiding the simulation-based pattern generation.

2.1 Logic-Simulation-Based Sequential ATPG

Simulation-based ATPG can be divided into two categories: (1) fault/error-simulation-based ATPG and (2) logic-simulation-based ATPG. In the former, the test generator dynamically collects the information related to fault/error detection during simulation; the information is fed back to guide the test generator. Since fault simulation can be costly, its application to large sequential circuits may be limited. On the other hand, instead of using fault coverage metrics, logic-simulation-based

test generators use characteristics embedded in the fault-free circuit to guide the search. Because there are no faults or design errors involved, logic-simulation based ATPGs have a much lower computational complexity and is not constrained to any underlying fault/error model.

Some logic-simulation-based sequential ATPGs have been proposed in the literature. In CRIS [10], the characteristic targeted by the logic-simulation-based ATPG is the number of toggles in the regions of fault-free circuits. The test generator tries to maximize the circuit activities and distribute the toggles evenly across the entire fault-free circuit, with the hope that more toggling activities can increase the chances for detecting hard-to-detect faults. Based on the observation that the test sequence generated by deterministic ATPGs generally traverses through a sequence of states, and the new states reached inside the sequence may help in improving fault coverage, in LOCSTEP [11], the targeted characteristic is the number of new states reached in the fault-free circuit. Because the number of state variables can be huge, maintaining and managing a global state table may be a challenge. In addition, the coverage may plateau early because not every globally new state is useful in raising the coverage. In [12], an attempt is made to differentiate among the set of new states reached via state partitioning. The ATPG tries to favor those states that are deemed more valuable by putting emphasis on those less explored state partitions. In [13, 14], enhanced state partitioning is proposed to discover the correlations among flip-flops (FFs) such that a good partition set is obtained.

2.1.1 NL-ATPG vs. PL-ATPG

Logic-simulation-based ATPGs have shown promise for generating effective tests for both manufacturing test and design validation. It has two sub-categories: (1) ATPG based on partitioned FSM traversal (PL-ATPG) and (2) ATPG based on non-partitioned FSM traversal (NL-ATPG). The difference between PL-ATPG and NL-ATPG resides mainly in the objective function on how to direct the search in the state space, shown in Figure 2.1. In NL-ATPG, the state variables are

not partitioned and the search tries to reach as many new global states as possible. On the other hand, in PL-ATPG, the state variables are partitioned into several groups, with each group given a different weight, (e.g., partitions consisted of FFs on the control path are given higher weights than partitions consisted of FFs on the data path), and the search will only consider a global state to be new and useful by the indications from the partitioned states. In other words, a combination of previously reached partial states is not regarded as a new and useful global state. To illustrate this concept, we represent a distinct 4-bit partial state with the hexadecimal notation; a "3" would represent a partial state "0011". For a state S of eight state variables, a couple (S_1, S_2) is needed to represent its entire value. For example, (5,A) represents global state "01011010". Assume that the current test set has reached the following global states: (0,0), (5,A), (2,3), (6,2). The partial states visited on S_1 and S_2 are thus $\{0,2,5,6\}$ and $\{0,2,3,A\}$, respectively. Consider the global state (2,2). It is a new global state, but is not considered new on the partitioned state space because it does not introduce new partial state to either S_1 or S_2.

Global State S

FF 1	FF 2	FF 3	FF 4	FF 5	FF 6	FF 7	FF 8

Partitioned States S1 and S2

FF 1	FF 2	FF 3	FF 4

FF 5	FF 6	FF 7	FF 8

Current State Table

S1:	S2 :
0	0
5	A
2	3
6	2

Example1

5	3
3	3

Example2

9	2
5	6

Figure 2.1. State Partition Examples

Based on this partitioned state information, consider the following two scenarios. First, suppose two candidate sequences drive the circuit to two new, different global states: (5,3) and (3,3). We can select only one sequence to be appended to the test set. If no distinction is made between these two global states, the search would simply picks one randomly because both are new with respect to the global state table. However, state partitioning can differentiate the two states by pointing out that (3,3) is more beneficial because it brings a new partial state in partition S_1, while (5,3) does not bring any new partial state in either partition.

Second, suppose the two different states reached by the two candidate sequences are (9,2) and (5,6), and again we can select only one sequence to be appended to the test set. As in the first scenario, both states are new global states. In addition, 9 is a new partial state on S_1, and 6 is a new partial state on S_2. By simply glancing these two global states, it would be difficult to favor one over the other. However, imposing different weights on different partitions makes it possible. A partition deemed to have greater influence on the circuit gets a greater weight. If the weight assigned to S_2 surpasses that of S_1, the search will pick the second candidate sequence to be appended to the test set.

From the above two examples, it is seen that PL-ATPG has the potential to be more efficient than NL-ATPG. We will delve into PL-ATPG in later chapters.

2.2 Logic BIST

BIST alleviates the burden of purchasing expensive Automatic-Test-Equipments (ATEs), whose prices are proportional to the number of pins on the tester. BIST is also a good alternative to perform at-speed [5] testing when the technology (clock rate, propagation delays, etc.) of ATEs lags behind that of CUT. Furthermore, BIST is a promising solution for testing embedded cores in the SOC environment, whose testabilities are greatly reduced due to limited accessibility.

Many BIST schemes have been proposed in literature. Pseudo-random test pattern generator (TPG) in the form of a linear-feedback-shift-register (LFSR) has been widely used. Weighted pseudo random TPG [15, 16] takes into consideration the subset of random-pattern-resistant faults to yield better results than random methods [15, 16, 17, 18]. The basic idea for weighted pseudo random is to bias the probability at each input based on the information gathered on the circuit. The weights can be obtained by using counter-based schemes [19], performing bit-fixing [20], or employing LFSR with good starting seed and feedback polynomial [21, 22, 23]. Many hardware pattern generators [18] often round-off optimal weights, hence producing patterns that are sub-optimal for certain circuits at a much lower computational cost. While (weighted) pseudo-random BIST architecture may not provide a satisfactory fault coverage in reasonable pattern length (especially for sequential circuit), deterministic BIST techniques attempt to solve this problem by storing deterministically-generated test patterns. However, the cost associated with the storage of the pre-calculated test patterns may be high if there is a large number of patterns.

2.2.1 Scan-based Design

To increase the testability of the CUT, scan-based designs have been proposed. The flip-flops in the CUT are modified to scan-flops by adding a multiplexer (MUX) before the data input pin to select between the data of the normal mode and that of the test mode. The design of a scan cell is shown in the upper half of Figure 2.2. The lower half of Figure 2.2 shows how scan cells are chained serially to form a scan chain in scan-based design. When $TESTMODE$ is low, the scan-flop acts as a normal flip-flop, and data from the combinational logic is picked by the MUX; when $TESTMODE$ is asserted, the scan-flop is acting in the test mode, and data from the test input is picked by the MUX. Scan-flops are chained together to form a *scan chain*. Each scan chain has a scan input pin to shift in the test vectors and a scan output pin to shift out the test responses. During pattern application, two phases are involved. In the *shifting phase*, $TESTMODE$ is high, data bits in the test vector are serially shifted in from the scan input pin, and test responses for the previous

vector are serially shifted out from the scan output pin. If the scan chain contains *N* scan-flops, the shifting phase would take *N* clock cycles. Then, in the *capture phase*, *TESTMODE* is set to low, and test responses for the current vector from the combinational logic would be captured into the scan-flops. The pattern application procedure for a scan-based design is shown in Figure 2.3, assuming the initial state of the CUT is unknown.

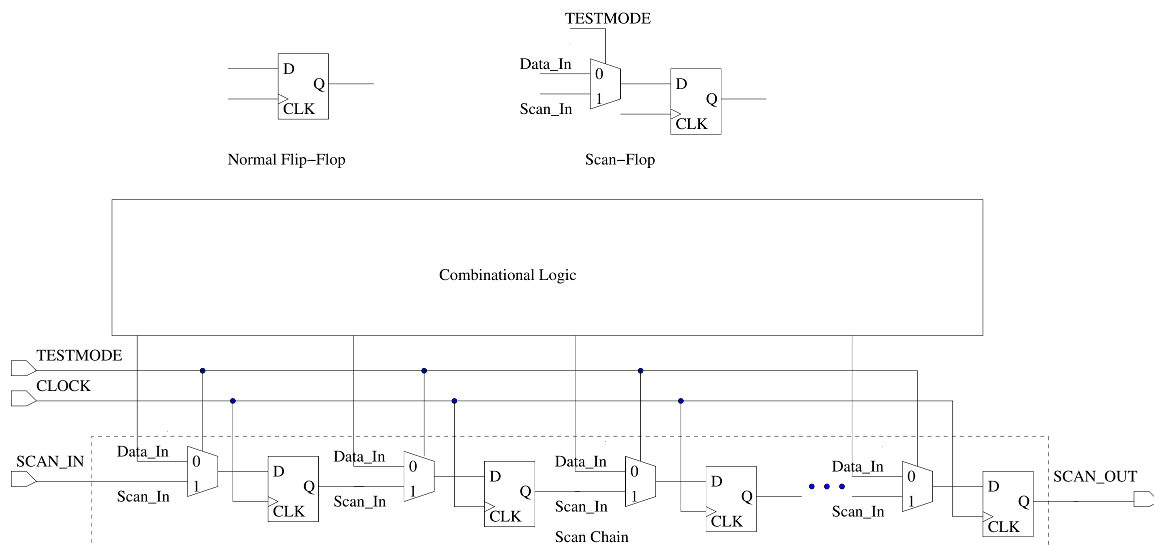


Figure 2.2. Scan Cell and Serial Scan Architecture

2.2.2 Low Power Schemes for Scan-based Logic BIST

While scan-based logic BIST greatly increases the test coverage of the CUT, the disadvantages with scan-based logic BIST are: (1) the prolonged test application time, and (2) the elevated test power. The serial scan architecture allows every flip-flop to be both controllable and observable via the two additional *SCAN_IN* and *SCAN_OUT* pins. As explained in the previous subsection, it takes *N*+1 cycles for a full-scan pattern to be applied, where *N* is the number of flip-flops in the scan chain. Among the *N*+1 cycles, *N* of them belong to the shifting phase, where significant ripplings may appear on the scan chain. The ripplings that originate from the scan chain can

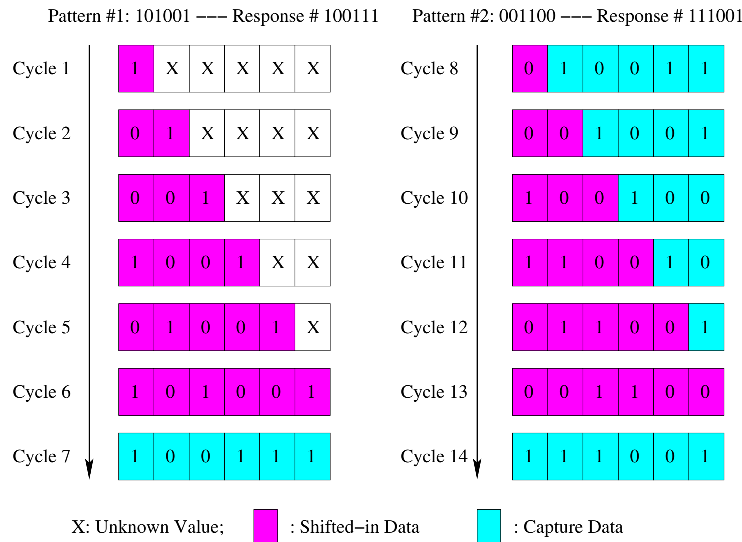


Figure 2.3. Pattern Application on Scan-based Design

reflect into more transitions inside the circuit, causing increased power dissipation on the circuit. Excessive power dissipation during testing can cause problems, many of which can permanently damage the CUT or make the CUT less reliable, e.g., elevated electromigration rate and increased power/ground noise. Thus power-dissipation issues are gaining more attentions in testing VLSI circuits.

The current low power schemes for scan-based logic BIST can be classified into 3 categories: (1) vector reordering, (2) scan chain reordering, and (3) clock gating. Some proposed low power BIST architectures are listed below. In [24], additional hardware for controlling ”pattern skip” and toggle suppression during the scan-out procedure is added to reduce power consumption. Although average power can be reduced to less than 10%, the hardware overhead due to the control logic needed can be very large. In [25] and [26], several heuristics are proposed to reorder test vectors and scan chain for full-scan sequential circuits. The algorithms need additional memory to store the reordered test vectors, which could bring in large hardware overhead. In [27], it partitions the primary and state inputs in BIST into two parts: one is fed by a normal-speed LFSR, while the

other is fed by a slow LFSR. This scheme has little hardware overhead and does not need memory to store any test pattern, but it cannot guarantee no loss of fault coverage. In [28], scan chain is modified by inserting logic between two neighboring flip-flops, the input patterns need to be modified correspondingly, and additional memory is needed to store the modified test patterns. A re-seeding and vector inhibiting technique [29] is proposed, which is another method to perform vector skip, in order to reduce power. In [30], a scheme to partition the scan chain into different groups is presented; together with vector reordering, some scan chains are disabled during testing without damaging fault coverage. Additional control logic and memory are needed for this technique. In [31], the scan chain is also partitioned into groups and only one of them is active for shifting at a time. Test power is thus reduced to approximately 1/N, where N is the number of groups. In [32], the ordinary scan architecture is modified to LT-RTPG (low-transition random test pattern generator) by adding a TFF (toggle flip-flop) and a k-input AND gate before the input of the scan chain to lower the switching activities inside the scan chain during shifting. Hardware overhead in this scheme is negligible and input patterns can be purely random, while there may be a little drop in fault coverage. In [33], a BIST architecture comprised of two TPGs: LT-RTPG and 3-weight WRBIST TPG is proposed. Random detectable faults and random resistant faults are separated and taken care of by LT-RTPG and WRBIST TPG respectively. The scheme can achieve 17-45% average power reduction and 100% fault efficiency at the cost of additional control logic. All the methods mentioned above focus on reducing average power dissipation. There also exist techniques [34, 35, 36] aiming at reducing the peak power, which we will not address in this dissertation. In [37], various low power testing techniques for VLSI circuits are surveyed with discussions on the opportunity to use each technique in varying situations.

2.3 Correlations

In the following subsections, we will explain the basic questions on:

1. What are correlations?

2. How to learn correlations?

3. How to use correlations?

2.3.1 Definition of Correlations

The word *correlation* has been defined in Merriam-Webster dictionary [38] as: a relation existing between phenomena or things or between mathematical or statistical variables which tend to vary, be associated, or occur together in a way not expected on the basis of chance alone. In probability theory and statistics, correlations indicate the direction and strength of the linear relationship between two random variables under certain conditions. Correlation measurements among variables are called *correlation coefficients*, which can be expressed by a *correlation matrix*. A *correlation matrix* C of n variables x_1, x_2, ..., and x_n is an $n \times n$ matrix, where the value of each entry records the correlation coefficient between the two corresponding variables. For example, $C[i][j]$ records the correlation coefficient between variable i and j.

It is meaningless to talk about correlations without specifying the context. Although there are some types of correlations which hold all the time universally, most of the interesting correlations only hold under certain conditions. The contents of the correlation matrices for the same set of variables under different conditions may differ a lot.

2.3.2 Examples of Correlations

Signal correlations are a very common phenomena in circuit behavior. The cause of signal correlations may be of the structure of the circuit. Static implications can be viewed as a type of structural correlations. Shown in Figure 2.4, for the two-input AND gate, c=1$\rightarrow\{a$=1, b=1$\}$. This correlation

holds as long as the type of the gate does not change.

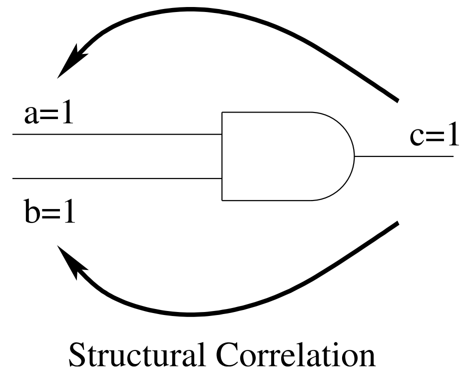


Figure 2.4. Correlations: Example I

Some correlations are statistics extracted from the data sample under certain conditions. An example is shown in Figure 2.5, where PI sequences for detecting some hard faults in the CUT are presented. Pin 0 is the global reset pin, which can initialize all the memory elements in the CUT to logic 0 when it is asserted. It is not desirable to have a lot of toggles on pin 0 as on other pins, because hard faults usually require a long sequence of states to be traversed before their detections and forcing the circuit to enter the reset state by asserting pin 0 can disrupt the useful sequence. For test sequences which can detect hard faults in the CUT, in most time frames, pin 0 will stay low. This temporal correlation on pin 0 (on the same variable, along different time frames) is the statistics extracted from the set of test sequences which can detect hard faults f_1, f_2 and f_3.

2.3.3 How to Learn Correlations

In this work, we focus on learning correlations from simulation. The correlations that we are interested in would be in three-fold: (1) beneficial for fault detection, (2) beneficial for property verification, and (3) beneficial for power reduction. The two types of correlations we have studied include: (1) temporal correlations, and (2) spatial correlations.

Time Frame	Test Seq f1	Test Seq f2	Test Seq f3
0	1 0 1 0	0 1 1 1	1 1 1 0
1	0 0 0 0	0 1 1 0	0 1 0 0
2	0 1 0 0	0 1 0 1	0 0 0 1
3	0 1 1 0	0 1 1 1	0 0 1 0
4	0 0 1 1		0 1 0 1
5			0 1 1 0
6			0 1 0 1

Figure 2.5. Correlations: Example II

1. *temporal correlations:* correlations on the same variable along different time frames;

2. *spatial correlations:* correlations among different variables at the same time frame;

The example shown in Figure 2.4 is on spatial correlation, and the example in Figure 2.5 is on temporal correlation.

2.3.4 How to Use Correlations

Correlations are proved to be very useful when they are employed in the pattern generation flow. The tests generated in the previous iteration are simulated and only those tests that appear to be helpful to the objective (fault detection, property verification or power minimization) may contain useful correlation information. Correlations extracted from the filtered tests will be embedded into the fitness/cost function such that tests generated in the current iteration will take those correlations into consideration. Iterations of test generation go on until the objective is achieved or some resources run out. The above flow can be described by the block diagram in Figure 2.6.

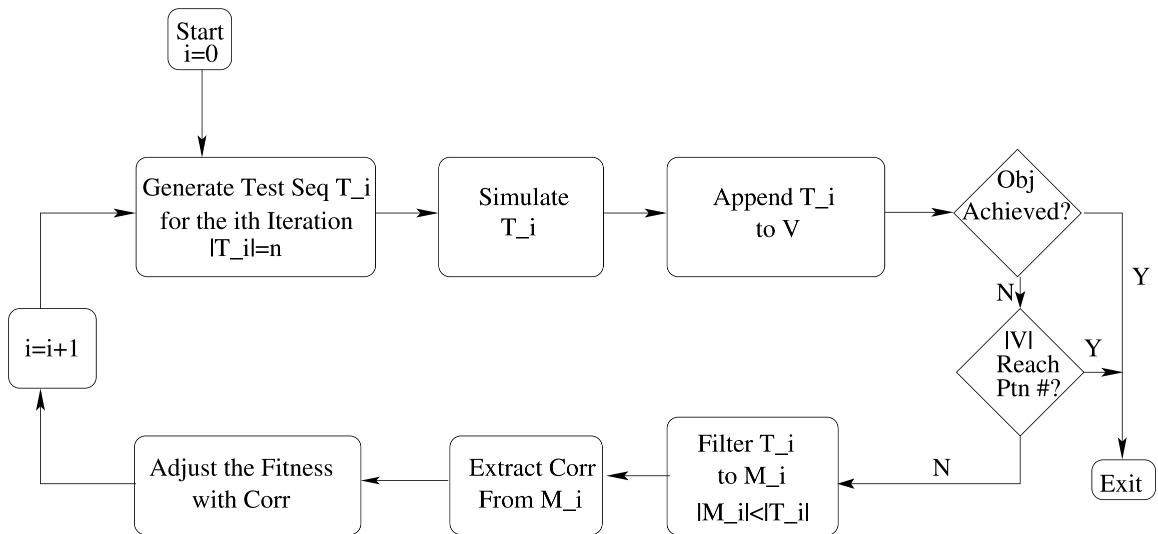


Figure 2.6. Employing Correlations into Test Generation

Chapter 3

Characteristic Faults and Spectral Information for Logic BIST

In this chapter, we investigate how the temporal correlations on PIs can be used in fault-simulation-based sequential ATPG. We present a new method of built-in-self-test for sequential circuits and system-on-a-chip using characteristic faults and circuit-specific spectral information in the form of one or more Hadamard coefficients. The Hadamard coefficients that contain the temporal correlations of PIs are extracted from the test sequences for a small set of characteristic faults of the circuit. By extracting a few characteristic faults from the circuit, we show that the detection of these characteristic faults is sufficient in detecting a vast majority of the remaining faults in the circuit. The small number of characteristic faults allows us to reduce the coefficients necessary for BIST. State relaxation is performed on the compacted test sequences to reduce the spectral noise further. Since we are targeting only a very small number of characteristic faults, the execution times for computing the spectra are greatly reduced. Our experimental results show that our new method can achieve high BIST coverage with both lower computational efforts and storage, with very few characteristic faults.

3.1 Introduction

Built-in-self-test (BIST) is playing an increasingly important role in VLSI testing. As the operating speed in modern VLSI technology increases, it becomes more difficult for test equipments to keep pace with the growing speed. Furthermore, BIST is a promising solution for testing embedded cores in the SOC environment, whose testabilities are greatly reduced due to limited accessibility.

An embedded controller or processor on an SOC makes it possible for the processing unit to generate patterns to test the rest of the chip [39, 40, 41, 42, 43]. This alleviates the need for additional hardware such as LFSRs to test the other embedded cores/peripherals, and by employing efficient algorithms, it is also possible to reduce storage.

In [12], temporal correlation was found to exist within test sets, and this information allows for the application of spectral techniques to ATPG [44]. In [43], spectrum-based BIST technique was proposed for SOCs with embedded processor. It generates vectors by using the spectral characteristic of each embedded core. To do this, it views each core as a digital system that is described by input-output system, described as functions of time. Any such function can be represented and reproduced in the time-domain, using its frequency-domain spectrum. The approach uses Hadamard transform to compute the spectral information of digital circuits. As noise-free signals are desired to capture the spectral characteristic of the signal, static test set compaction is employed to remove any unnecessary vectors and thus filter some of the unwanted noise. As a result, a test sequence can be described by the corresponding spectral coefficients, and it preserves the fault detection criterion. At this point, self-testing becomes the problem of determining the spectrum of the circuit under test (CUT).

In this chapter, we address the problem of refining the spectrum of the CUT, by picking a small set of *"characteristic faults"* from the circuit and employing state relaxation. Unlike [43], where spectral information is extracted from the entire fault set, only a very small number of the faults is

used in this work. In essence, the set of characteristic faults can be viewed as representative faults of the entire circuit. In [45], test cubes are computed for hard faults such that many other easy faults can be detected simultaneously; however, only combinational circuits are considered. State relaxation [46] is performed on the compacted test sequences of the characteristic faults to further reduce noise. We present efficient algorithms on selecting characteristic faults.

The remainder of the chapter is organized as follows. Section 3.2 gives an overview and motivation for the new spectral BIST. Section 3.3 explains the algorithm of computing characteristic faults and its application to BIST. Section 3.4 discusses the details of state relaxation. Section 3.5 reports the experimental results, and Section 3.6 summarizes the chapter.

3.2 Overview and Motivation

Previously, spectrum-based BIST [43] has shown the important role spectral information can play. The spectral information it uses is extracted from the compacted test set for all detected faults. Because wider spectra contain more noise, pruning the spectra to rid the noise will help. Generally, sequences that detect hard faults frequently can detect many other faults. Based on this observation, we propose to focus on obtaining the spectral information for only the more difficult faults.

Our work begins with the test set that detects only the k latest detected faults. We perform relaxation on this test set to relax any unnecessary input bits to don't-care values (X's); the relaxed test set still ensures detection of the k faults. The relaxed sequence essentially removes additional noise from the original sequence, and the non-relaxed input bits in the relaxed sequence form the basic spectrum and are sufficient to traverse the state space such that the k hard faults are still detected. Surprisingly, many of the input bits can be relaxed. The states traversed may not be fully specified due to the don't cares in the input sequence. Furthermore, if detection of these k faults also detects a vast majority of the remaining faults, the derived spectrum can be viewed as the representative spectrum for the entire circuit. Following this assumption, our goal is to extract the spectrum from

a *very small fault set*, and this spectrum is able to represent the characteristic of the circuit. As in [43], we employ signal processing techniques to extract the spectral information and generate spectral BIST vectors. In addition, in our scheme, state relaxation is performed on the compacted test set for the *k* faults to reduce noise prior to extraction of spectral information.

The overall framework for extracting the spectral characteristics is presented in [43], however, characteristic faults were not identified. Shown in Figure 3.1, in this work, by starting with random vectors, we first *filter* the random test set using static compaction for the *k* latest detected faults (not the whole detected fault set). Then, state relaxation is applied to the compacted test sequence to further reduce any additional embedded noise. The spectral coefficients are extracted from this relaxed, compacted test sequence through Hadamard transform, where the predominant Hadamard components are identified. New vectors are added to the test set based on the extracted spectrum. This process iterates until the stopping criterion is reached. At the end of this process, only the spectral characteristics for the *k* faults in the iteration where highest fault coverage was reached are collected and stored, and those *k* faults are called *characteristic faults*. The final spectrum is used to generate BIST vectors in the SOC. Note that in our process, the spectral coefficients from one iteration will be used by BIST. This is different from [43], where the coefficients from an iteration are appended to the final set if there is an increase in fault coverage.

3.3 Computation of Characteristic Faults

Instead of targeting the entire fault list to extract spectral information, we want to find "characteristic" faults within the fault set. Characteristic faults should have the property that a test set which detects them is also able to detect many other faults in the circuit. For this reason, the spectrum corresponding to the characteristic fault set is a representative spectrum for all the faults.

Because sequences that detect hard faults generally detects many other faults as well, our aim is on identifying the set of hard faults such that detecting them will maximize detection of the

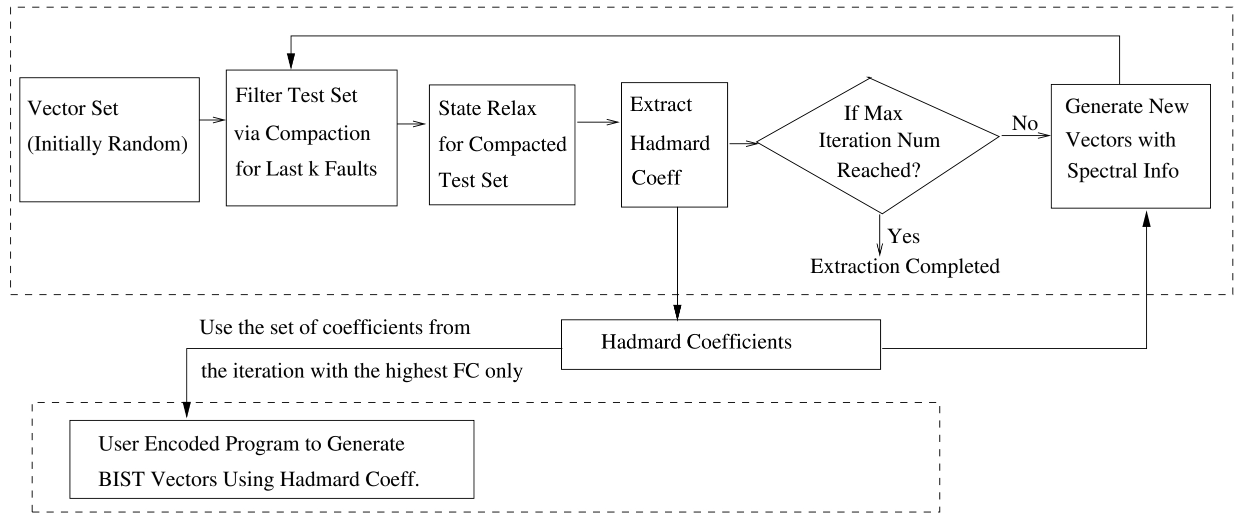


Figure 3.1. Overall Framework

rest of the faults in the circuit. To do so, we modify the spectral ATPG process [43] in which the corresponding target faults in each iteration are associated with the derived spectrum. For instance, when the spectrum is extracted for the k last-detected faults in iteration i, the new vectors to be generated are simulated and the set of faults, S_i detected by these new vectors are recorded. The set S_i indicates the set of faults that have similar spectral characteristics as the k representative faults. In the next iteration, we would likewise compute S_{i+1}. The k faults in the jth iteration corresponding to the largest fault set S_j is chosen to be the characteristic fault set.

The characteristic-fault extraction algorithm is outlined in Figure 3.2. The algorithm is simple to follow and we will show in experiments that it is also efficient in getting the characteristic fault set that achieves high fault coverages.

T_0 = random test vectors;
$i = done = S_{max} = 0$;
while (not *done*) and (i<max_iteration)
    fault simulate T_i;
    K_i = k last-detected faults;
    $C_i = T_i$ filtered for K_i;
    $R_i = C_i$ relaxed;
    perform Hadamard Transform on R_i to obtain spectral coefficients H_i;
    E_i = new vectors generated using coeff. H_i;
    S_i = faults detected by E_i;
    if $S_i > S_{max}$
        characteristic fault set = K_i;
        $S_{max} = S_i$;
    i++;
    if no improvements in 3 consecutive iterations
        done = 1;

Figure 3.2. Algorithm for Extracting Characteristic Faults

3.4 State Relaxation

3.4.1 Review of Support Sets

A test vector is *fully specified* if all inputs are specified to 0 or 1 (*i.e.,* no input assumes a value of X). Test vectors that contain X's are said to be *partially specified*. A *support set* (SS) for a primary output Z is any set of signals (including primary inputs) that satisfy all the following conditions:

1. All signals in the set assume a logic value 0 or 1.
2. The primary output Z is a member of the set.
3. The logic value on any signal (except PIs) in the support set is uniquely determined by values of other signals in the SS.

In the case of multiple primary outputs, condition 2 is modified to require that each of the POs be included in the support set. Likewise, support sets can also be computed for sequential circuits, where condition 2 is modified to include any next state variable v that satisfies the following condition:

- Next state variable v that is at a logic value 0 or 1.

The *support signals* for a gate are the smallest subset of required signal that uniquely determine the current logic value of the gate. For example, consider an AND gate a that has two inputs b and c. Suppose a=1, then both b, c must be 1 and be support signals for a. If a=0, b=1, and c =0, then support signal is simply b since the value on a is uniquely determined by signal b. In case that a=0, b=0, c=0, b, c are at level v(b) and v(c) respectively, and v(b)>v(c), we will use the following criteria:

- If one of the possible support signals of the gate has already been included in the support set of the circuit, then this signal is selected as the support signal of the gate;
- Otherwise, we choose a support signal at the lowest level;

In this case, if b is already in the support set of the circuit, then b will be the only support signal for a; otherwise, c will be the support signal of a because c is at a lower level.

A support set is *irredundant* if no signal in the set can be deleted without violating conditions 2 or 3. A *minimum* support set has the least cardinality among all possible support sets. It is desirable to compute a support set for PIs with small cardinality as this leads to a *cleaner* signal for later processing. However, attempting to compute the minimum support set for each input vector is computationally expensive.

By computing the support set for each time-frame of sequential circuit, both the intermediate state and input sequence can go from fully specified to partially specified. The procedure [46] in Figure 3.3 efficiently computes an irredundant support set, and takes a list of gates L with known logic values. In the above procedure, we assume that the circuit is levelized and the input vectors have been simulated to determine the value of each gate in the circuit. The support sets are, then, computed based on the logic values for each corresponding vector.

3.4.2 Relaxation in Sequential Circuit

For sequential circuits, we perform state relaxation in a reversed order, that is, starting from the last pattern. Let's consider benchmark circuit s27 [47] with gate 10 stuck at 0. The original compacted test set $S1$ for detecting this fault is {1011, 1001, 0101, 0111, 0011, 0110}, where the fault is detected by the last vector.

Gate values for both good and faulty circuits are shown in Figures 3.4 and 3.5 respectively for

```
Procedure COMPUTE_SUPPORT_SET( L )
    support set S = L;
    while (∃ unsupported gates in L)
        g = unsupported gate in L with maximal level;
        ss = minimal support signals for g;
        add support signals ss to S;
        for all unsupported gates i in ss
            add i to L;
        mark g supported;
    return support set S;
```

Figure 3.3. Algorithm for Computing Minimal Support Set

vectors 5 and 4. The values are shown in the form of [good/faulty]. Starting from pattern 5, the constraint list L contains only the PO which differentiates good circuit and faulty circuit; since pattern #6 (time frame 5) is the last vector, no next state FF is included in L. In this particular case, the fault is detected by pattern 5 on output gate 18, with good circuit of logic 0 and faulty circuit of logic 1. Thus, L for this vector is simply $\{18\}$. Calling procedure COMPUTE_SUPPORT_SET() with this constraint list L, we obtain the support set for good and faulty circuit respectively, listed below in the levelized fashion. The value ”12[1/0]” indicates that gate # 12 = logic 1 is included as part of support set for the fault-free circuit, while gate 12 with logic 0 is included in the support set of the faulty circuit.

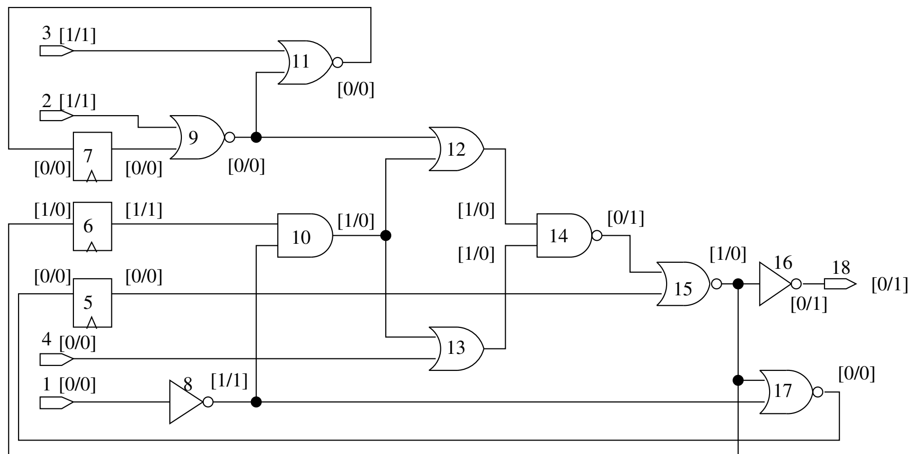


Figure 3.4. Good/Faulty Circuit Values at Time Frame 5

```
Support Set for Good and Faulty Circuits, vec #5
0  5[0/0] 6[1/1] 1[0/0] 4[X/0] 2[X/1]
1  8[1/1] 9[X/0]
2 10[1/1]
3 12[1/0] 13[1/0]
4 14[0/1]
5 15[1/0]
6 16[0/1]
7 18[0/1]
```

We combine the support sets for good and faulty circuits at level 0 and split that into two lists $L1$ and $L2$ by gate type. $L1$ consists only PIs and $L2$ contains only FFs. PIs and FFs which are not contained in either $L1$ or $L2$ can be set to X. By doing this, we have relaxed the input and state for the current time frame. In this example, $L1$ is $\{1\ 4\ 2\}$ and $L2$ is $\{5\ 6\}$. $L2$ will be used to form the constraint list for the preceding pattern. We move on to relax pattern 4 the same way by calling the procedure, and this time, constraint list L will be the list of $L2$, which is $\{5\ 6\}$. Computing the support set the same way and we get the following support sets of pattern 4 for good and faulty circuit respectively:

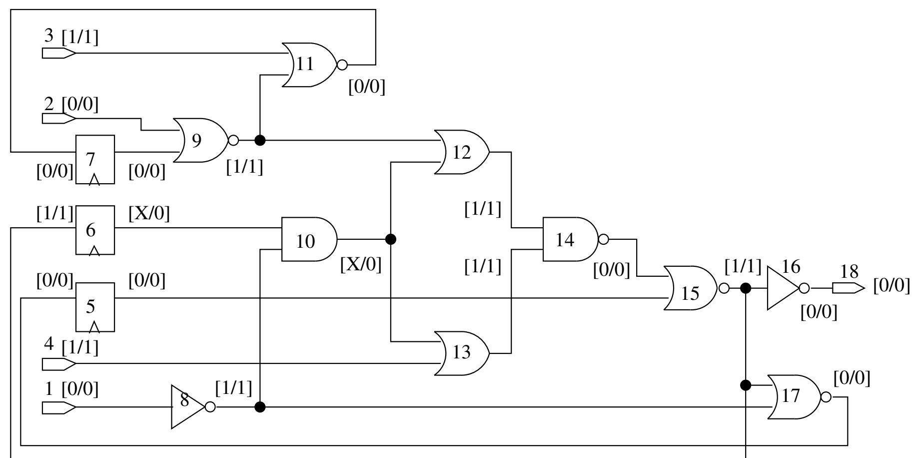


Figure 3.5. Good/Faulty Circuit Values at Time Frame 4

```
Support Set for Good and Faulty Circuits, vec #4
0  5[0/0] 4[1/1] 7[0/0] 2[0/0]
1  9[1/1]
2 12[1/1] 13[1/1]
3 14[0/0]
4 15[1/1]
5 17[0/0]
```

The above procedure is repeated until we've reached the first vector. in reversed order. Shown below are the relaxed inputs (from input 1 to 4 in order) and intermediate states(from FF 5 to 7 in order) for our s27 example:

Vec \#	Original Input	Relaxed Input	Relaxed State
0	1011	XXXX	XXX
1	1001	XXXX	XXX
2	0101	XXXX	XXX
3	0111	0X1X	XXX
4	0011	X0X1	0X0
5	0110	01X0	010

3.5 Experimental Results

We conducted experiments of our approach on both ISCAS89 [47] and ITC99 [48] benchmark circuits. After computing the characteristic faults for each circuit, we allowed for a maximum of 70,000 vectors from the spectral information extracted for the characteristic faults after state relaxation. When performing the Hadamard transform, a relaxed '*X*' does not contribute to the spectrum extraction. Recall that in [43], the final coefficient set stored into the SOC is a concatenation of coefficient sets from different iterations. Once a new fault is detected in the iteration, the coefficient extracted from the whole compacted test set of that iteration will be included into the final coefficient set. That may result in larger storage requirement. We will show later in Table 3.3 how the two schemes differ in both computational effort and storage requirements. For all circuits, the number of iterations is set to 20.

Table 3.1 compares the results among STRATEGATE [49], weighted random BIST, spectral method [43] and our technique. Note that **only one** characteristic fault is used for our technique. In this table, the total number of faults is first listed for each circuit, followed by the coverage achieved by STRATEGATE. Next, BIST coverages are reported for ideal weighted random, rounded-off weighted random, [43], and finally our approach based on characteristic faults. For instance, in circuit s5378, STRATEGATE generated a test set that detected 3639 faults, ideal weighted random BIST detected 3127 faults, rounded-off weighted random BIST detected 3083 faults, 3596 faults were detected by [43], and we detected 3611 faults with spectrum for **only one** characteristic fault! Likewise, in circuit b12, our technique detected 1648 faults with spectral information using only one characteristic fault, while STRATEGATE, ideal, rounded-off weighted random BIST, and [43] detected 1488, 663, 636 and 1621 faults, respectively.

From this table, we can see that with only one characteristic fault, our fault coverages are very close to those obtained by state-of-the-art sequential ATPG for most circuits. In all circuits except for s1196 and s1238, the results of our technique either surpass or equal the results obtained for

Table 3.1. Comparison of Fault Coverages

Ckt	Total Faults	# Faults Detected by STRATEGATE [49]	# Faults Detected by Non-Scan BIST			
			Weighted-Random Patterns		Spectral Patterns	
			Ideal Weights	Rounded-off Weights	[43]	Ours
s382	399	**364**	329	116	**364**	357
s400	428	**384**	306	106	**384**	376
s526	555	**454**	95	94	**454**	442
s713	581	**476**	476	476	**476**	**476**
s1196	1242	**1239**	1233	1228	*1237*	1227
s1238	1355	**1282**	1276	1270	**1282**	1266
s1423	1515	**1414**	1319	1167	**1414**	**1414**
s1488	1486	**1444**	1442	1410	**1444**	**1444**
s1494	1506	**1453**	1451	1418	**1453**	**1453**
s5378	4603	**3639**	3127	3083	3596	*3611*
b01	135	**133**	**133**	**133**	**133**	**133**
b04	1346	**1168**	**1168**	**1168**	**1168**	**1168**
b08	489	**463**	461	438	**463**	**463**
b11	1089	1003	937	898	**1004**	**1004**
b12	3102	1488	663	636	1621	**1648**

Only **one** characteristic fault is used under ”Ours” column

the ideal weighted random technique. For some hard-to-test circuits such as s5378 and b12, our technique is able to detect significantly more faults than the weighted random BIST approach and also outperforms [43], indicating that the spectra extracted by our new scheme are more efficient. The reason our approach did not perform as well in s1196 and s1238 is that these two circuits are randomly testable and thus the size of the compacted test set for a single characteristic fault is too small, typically only 5 vectors. Due to this reason, spectral information for the entire circuit cannot be fully captured by these few vectors.

To increase fault coverage, the immediate thing to do is to increase the number of characteristic faults. For those circuits which saw a loss in fault coverage with respect to [43], we increased the size of characteristic fault set, k, gradually until the fault coverage reaches that of STRATEGATE. The results are reported for increasing number of characteristic faults in Table 3.2. Once the fault coverage reached a desired level, we discontinue increasing the number of characteristic faults, and a "/" is placed under that column. In circuits s1196 and s1238, both of which are randomly testable, by increasing the number of characteristic faults, the fault coverages improved significantly as expected. For many other circuits, increasing the number of characteristic faults did not improve the results, since the single characteristic fault already achieved very high fault coverages.

Table 3.3 reports the speedup and storage reduction between our technique and [43]. The time reported is in seconds and the storage reported is the size of the final spectral coefficient set. Take s382 for example. The execution time for [43] and ours are 261 seconds and 212 seconds respectively. The speedup is due to the fact that we are filtering/compacting for the characteristic faults only. The number of spectral coefficients in our approach is 58, while the size by [43] would require 567. For this small circuit, nearly 1 order of magnitude reduction in storage is achieved. Results for s5378 showed that more than 2 orders of magnitude reduction in storage is achieved, with only 24 spectral coefficients needed for the circuit.

Table 3.2. Fault Coverage for Varying Number of Characteristic Faults

Circuit	1 Chara Fault	4 Chara Faults	6 Chara Faults	10 Chara Faults	14 Chara Faults	16 Chara Faults
s382	357	357	364	/	/	/
s400	376	384	/	/	/	/
s526	442	444	452	/	/	/
s1196	1227	1230	1233	1237	/	/
s1238	1266	1273	1276	1280	1281	1282

Table 3.3. Computational Cost and Storage Requirement

Circuit	Execution Times			Storage Requirements		
	[43] Time(s)	Our Time(s)	Time Speedup	[43] Storage	Our Storage	Storage Reduction
s382	261	212	1.23	567	58	9.78
s400	267	216	1.24	567	75	7.56
s526	475	254	1.87	1106	62	17.84
s713	297	191	1.55	89	4	22.25
s1196	124	101	1.23	232	5	46.4
s1238	189	142	1.33	243	5	48.6
s1423	17400	1345	12.94	14421	34	424.15
s1488	731	435	1.68	413	20	20.65
s1494	768	514	1.49	413	20	20.65
s5378	7085	4841	1.46	3426	24	142.75
b01	66	54	1.22	62	8	7.75
b04	659	462	1.43	158	8	19.75
b08	193	145	1.33	428	21	20.40
b11	594	354	1.68	2003	90	22.26
b12	20700	3631	5.70	12866	115	111.88

3.6 Summary

Temporal correlations on PIs were investigated in this chapter. We presented an effective approach for logic BIST using characteristic faults and spectrum information. We demonstrated that many faults in a circuit share similar spectral characteristics, which can be captured by a small number of characteristic faults. By intelligently selecting a small set of characteristic faults, we are able to spectrally characterize the circuit, and the extracted temporal correlations are sufficient in aiding BIST to obtain extremely high fault coverages. Our technique achieves the same or higher fault coverage than previously proposed BIST approaches, while the computational effort and storage needed are much less since our target changes from the total detected fault set to only a few target faults.

Chapter 4

Testing Embedded Sequential Cores in Parallel Using Spectrum-Based BIST

As an extension of the work in Chapter 3, in this chapter, we continue investigating the role temporal correlations play in helping sequential ATPGs. We present a new BIST architecture for system-on-a-chip by employing the temporal correlations on PIs, which can test a cluster of embedded sequential cores simultaneously. The compressed spectrum which contains the temporal correlations for a cluster of cores under test is computed by performing spectral analysis individually on all cores. Because there is no need to combine the cores to extract the spectrum for the entire cluster, the computation complexity is greatly reduced. For each individual core, we propose an interleaved state relaxation on the compacted test sequence for its characteristic fault set, leading to a partially specified, interleaved sequence which can be merged in a much easier way. A delay network and a switching network are added selectively to allow for more aggressive merging of spectra. Experimental results show that the same level of fault coverage can be achieved for each individual core with negligible hardware overhead, while the test application time for testing the entire cluster can be reduced by up to 4 times, and the test data storage requirement is reduced by up to 42%.

4.1 Introduction

In a system-on-a-chip environment where testabilities are greatly reduced due to limited access of the cores, it becomes undesirable to test each embedded sequential core in a serial manner, as the total test application time for sequential testing can become prohibitively long. Moreover, the area overhead of linear-feedback-shift-register (LFSR) based BIST circuitry to each embedded core can consume a large chip real estate. These problems present opportunities for designing a BIST framework for testing multiple embedded cores in parallel, reducing both the test application time and the area overhead.

An integrated technique for extensive optimization of the final test solution for SOC using *simulated annealing* has been proposed in [50]. An embedded controller or processor on an SOC makes it possible for the processing unit to generate patterns to test the rest of the chip [39, 40, 41, 42, 43].

Based on the work in Chapter 3, this chapter addresses the problem of reducing the test application time by testing a cluster of sequential cores on the SOC simultaneously, instead of testing each one individually and separately. (In the rest of the chapter, we will simply use "core" to refer to sequential cores.) Rather than storing separate spectra for individual cores as in Chapter 3, a single compressed spectrum for all the cores inside the cluster is extracted from the *combined* input sequence from each single core. The final spectrum should contain either the exact spectrum of an individual core or only allow minor changes to the spectrum of a core. The computational effort for getting the spectrum is on the same order as for a single core, since we do not first combine all the cores for spectral analysis. The storage requirement can be reduced up to 42% in our experiments, (the maximal storage reduction achieved in ideal cases can be 50%), although some spectral information for a single core may be lost. We call the above procedure *spectral compression*, and the spectrum after combining is termed *compressed spectrum*. By using the *compressed spectrum* for a cluster of cores, a set of cores can be tested as a single unit so that test application time is expected to be reduced. The test quality for the cluster of cores will largely rely

on the quality of the *compressed spectrum*. A *compressed spectrum* with less information loss is considered to be better. To produce a better *compressed spectrum*, *interleaved state relaxation* is performed on the compacted test sequence of the characteristic fault set for each individual core; also some DFT (design for testability), including a delay network and a switching network, is added selectively before the inputs of each core. With our framework, the test application time for testing the entire cluster can be reduced by up to 4 times.

The rest of the chapter begins with Section 4.2 giving an overview and motivation for the new spectral BIST. Section 4.3 presents the details of *interleaved state relaxation* and DFT insertion. Section 4.4 reports experimental results, and Section 4.5 summarizes the chapter.

4.2 Overview and Motivation

In the SOC environment, the test access mechanism (TAM) is used to deliver test stimuli from the source (which generates test stimuli) to cores and also to deliver responses from cores to the sink (which evaluates test responses). From the literature [51], the proposed TAM architectures can be classified into four categories:

1. Multiplexing: methods in this category [52] either directly multiplex the test pins to the PIs/POs (primary outputs), or modify the cores such that each core has a transparent mode for testing. The problems associated with it are non-scalability of the architecture, large overhead area, long test time and limited scope of use for complex SOCs;

2. Serial connecting: methods in this category [53, 54] use the IEEE 1149.1 standard. It has the disadvantage that as the number and complexity of the cores increase, a serial solution will be very costly in test time;

3. Indirect access: methods in this category emphasize on modularity, generality, and configurability of the architecture to exploit the advantages offered by the reuse paradigm, such as

NIMA proposed in [55];

4. Bus-based connection: methods in this category are most efficient in terms of trading-off increased overhead area for reduced test access time. Proposed schemes include [56, 57];

In this chapter, we propose a new bus-based TAM architecture for efficiently testing sequential cores on the SOC in parallel, using spectrum-based BIST. Previously, spectrum-based BIST [6] has shown the effectiveness spectral information can play in generating effective vectors. We made the observation that after *state relaxation*, a large percentage of the relaxed sequence, from which the spectrum is extracted, is don't care (X). In this chapter, we further develop this concept by performing *interleaved state relaxation* to interleave the specified bits in the input bit stream, simplifying the combining of the interleaved relaxed sequences of several cores. The cores are divided into two sets, and we name them "base set" and "target set". Bit streams of the sequences in the target set are to be superimposed onto those in the base set. Input sequences in the same set are concatenated, while input sequences from different sets are merged similar to crossing our fingers. Figure 4.1 shows how *interleaved state relaxation* helps in performing better merging of bit streams. The letters above the blocks (e.g., $A1$, $B2$) indicate the input index of each core. A conflict arises when we try to combine two specified bits with opposing logic values. Five possible conflicts can arise after merging the two relaxed bit streams on the left column; on the other hand, by employing our interleaved state relaxation, no conflicts arise after merging.

Spectral analysis is performed on the combined sequence, and the extracted spectrum can be treated as the spectrum for all cores in the cluster, which contains either the exact or a very close replica of the original spectrum of each core. In this way, a cluster of cores can be tested simultaneously using a single spectrum. The test application time for testing all cores in a cluster using the final spectrum will not exceed the time needed for testing the cores sequentially, and the storage requirement for testing the cluster of cores can be reduced correspondingly. There may be a slight penalty: the spectrum of each single core may not be fully recovered due to the merging of the

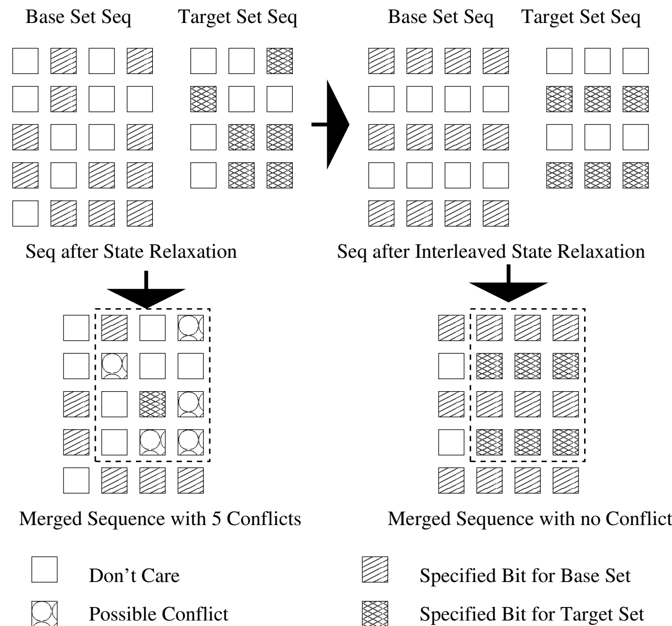


Figure 4.1. Interleaved State Relaxation

spectra. Because there is information loss, we call our procedure *spectral compression* and the extracted spectrum for the cluster of cores is so called *compressed spectrum*.

Figure 4.2 shows the proposed BIST architecture. The cores on the SOC are first divided into clusters (based on region, power consumption constraints, etc.), and each cluster is to be tested as a whole. There is a test bus on the SOC to deliver test patterns to each cluster. Between the test bus and the cluster-under-test, we may need to insert a delay network to supply different amount of delay for different inputs. The internal connections between delay units inside the delay network decide which and how inputs to different cores share the same spectral information, which can be considered as a switching network. Figure 4.3 further explains the reason behind the *delay network* and *switch network* as they play an important role in the final quality of the *compressed spectrum*. On the left side of Figure 4.3, five possible conflicts may arise when trying to merge two interleaved sequences; by employing a unit delay on the 2nd input (B2) of the target set sequence, and switching the 3rd input (B3) of the target sequence to the 1st input (A1) of the base sequence

(previously with the 4th input (A4) of base sequence), all conflicts are removed.

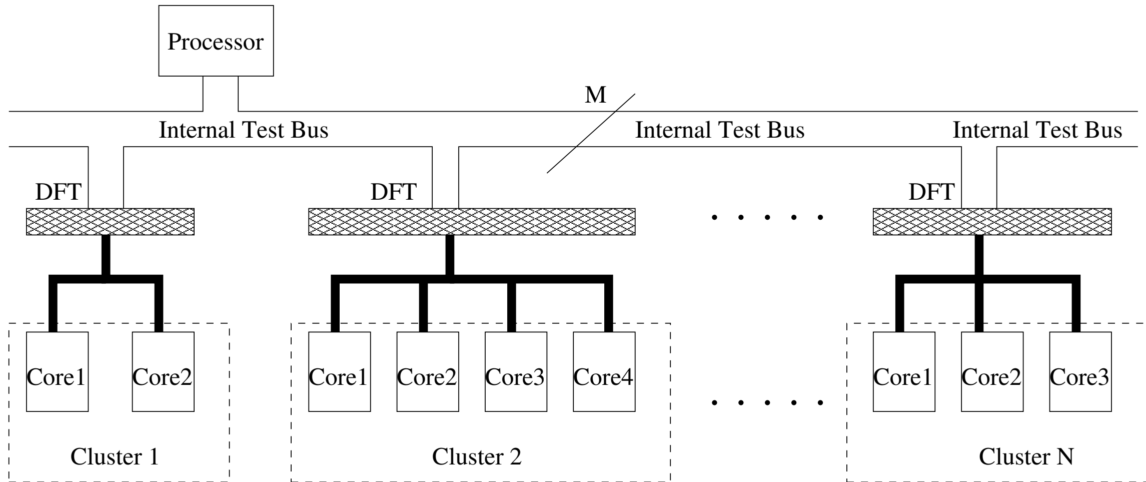


Figure 4.2. BIST Architecture

4.3 Interleaved State Relaxation & DFT Insertion

4.3.1 BIST Architecture

We will first explain how several cores can be tested together. The BIST architecture used for testing the cores in parallel is shown in Figure 4.4. It is composed of an m-bit ($m = 3$ in our example) wide test bus $A0$, a phase selection network $A1$, a delay network $A2$. $A4$ is the cluster of cores under testing, and $A3$ is a multiplexer network before each PI of each core for selecting test input and functional input. Assume that we have divided the cores on the SOC into N clusters. Cluster i consists of two cores, core$A$ and core$B$, shown in the figure. $A1$ is needed when the number of PIs of any of the cores exceeds the width of the test bus, and it decides whether to bypass the data on the test bus or hold the previous value. Inside $A1$ are a set of multiplexers controlled by the phase selection signals Pi and the test enable signal Ti for cluster i. If the maximal number of

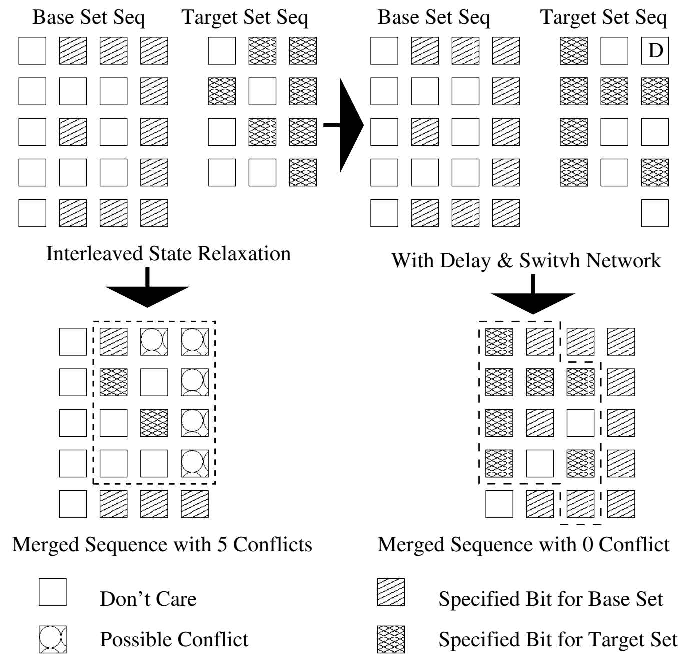


Figure 4.3. Using Delay Unit and Switching to Reduce Conflicts

PIs for a single core inside a cluster is M ($M = 4$ in our example), then it takes at least $[M/m]+1$ clock cycles to load a pattern, thus phase selection signal Pi will be of $log_2([M/m]+1)$ bit wide. For example, to load a test pattern for testing core A and core B in cluster i in Figure 4.3, it takes $[4/3]+1 = 2$ clock cycles, and the phase selection signal Pi is of $log_2([4/3]+1) = 1$ bit wide. When Ti and Pi are both asserted, the values on the test bus are selected to go through $A1$, while when Pi is not asserted, the values feeding into $A2$ keep the same as those in the previous clock cycle. In Table 4.1, $S0(1)$ represents the value of the test bus on bit $S0$, in time frame 1. Other notations can be explained the same way. Table 4.1 shows that in cycle 1, signals $J0$, $J1$ and $J2$ get their corresponding values from the test bus. Then in cycle 2, these signals hold their previous values, and signal $J3$ gets its value from the test bus.

Inside $A2$ is a set of delay units. As we saw before in Figure 4.3, different inputs of the same core may get misaligned during merging to prevent spectral information from losing and $A2$ is used to

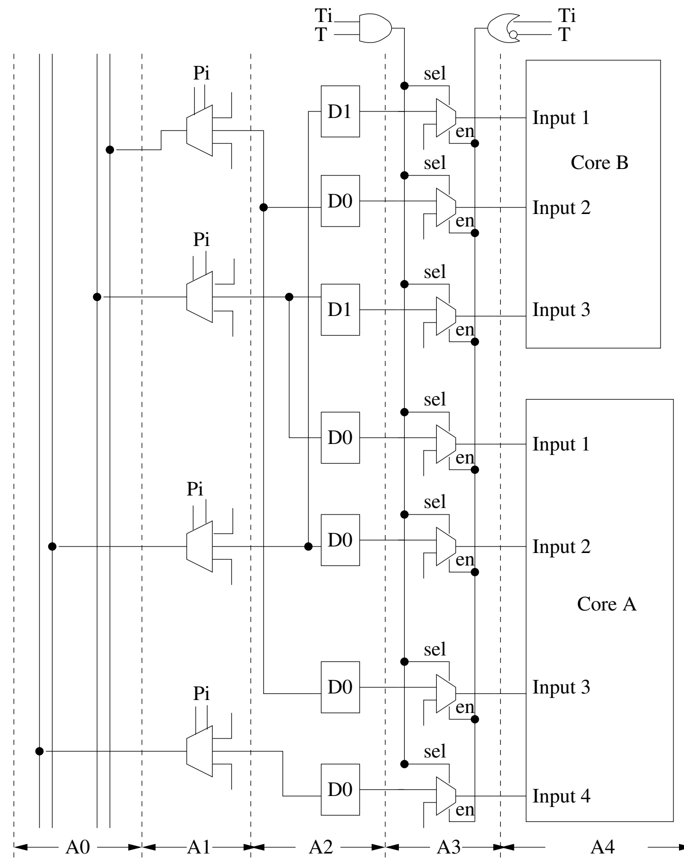


Figure 4.4. Delay Network and Internal Connections

resynchronize the inputs of the same core. The notation "$D1$" represents a unit which can provide delay of a single clock cycle, in which the number indicates the amount of delay. Simply, "$D0$" indicates no delay, and no extra hardware is needed. $A3$ is used to select input value while the core is in different mode. The mode of the cluster is controlled by the test enable signal T of the chip and the test enable signal Ti for cluster i, shown in Table 4.2. Note that $A0$, $A1$ and $A3$ are also included as necessary parts in other BIST architecture. The only extra hardware introduced is $A2$, whose size can be bounded by restricting the maximal delay value.

Figure 4.5 shows an example for configuring the delay network and the switching network when the maximal amount of delay allowed is 1. To reduce conflicts, three input pairs (A2, B1, 0), (A3,

Table 4.1. Applying a Single Test Pattern for Cluster i

cycle	Ti	$Pi(0)$	Data			
			$J0$	$J1$	$J2$	$J3$
1	1	1	$S0(1)$	$S1(1)$	$S2(1)$	X
2	1	0	$S0(1)$	$S1(1)$	$S2(1)$	$S0(2)$

Table 4.2. Different Modes for the Chip and a Cluster

T	Ti	Mode	
		Chip	Cluster i
0	X	Functional Mode	Functional Mode
1	0	Testing Mode	Hold Mode
1	1	Testing Mode	Testing Mode

B2, 1), (A1, B3, 0) are formed, where the items inside the brackets are input index of the base core, input index of the target core, and inserted time frame correspondingly. For example, (A3, B2, 1) means input #2 of core B is inserted at time frame 1 into input #3 of core A. The delay network and its internal connections are shown as $A2$ in Figure 4.4.

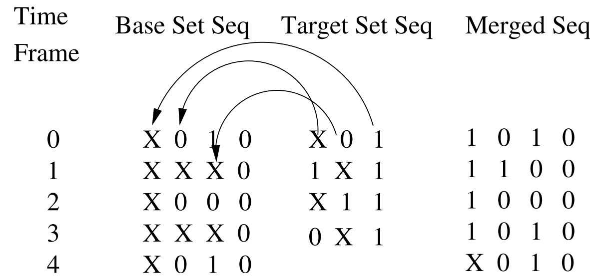


Figure 4.5. A Merging Example with Delay and Switching Network

4.3.2 Additional Terminologies

After state relaxation, for partially specified bit stream of length N:

Freedom f_i is defined as the ratio of the number of don't cares in the bit stream for the ith input to the sequence length N, shown in Equation (4.1). For example, if the bit stream for input i is "0X0X1X", then N is 6 and the number of don't cares is 3. Hence, f_i is 3/6=0.5. *Freedom* for input i shows how important that input plays in the spectrum. Input with a larger *freedom* is less important and has more room, while input with a smaller value of *freedom* contains more spectral information with less room. We prefer to form inputs with larger *freedoms* in merging.

$$f_i = \frac{\#\, of\ don't\ cares\ for\ input\ i}{sequence\ length\ N} \tag{4.1}$$

Rate of interleaf l_i is defined in Equation (4.2) as 1 minus the ratio of non-interleaved bit pairs in the bit stream for the ith input to the sequence length N. A non-interleaved bit pair is 2 consecutively specified bits along the time axis for a certain input. For example, if the bit stream for input i is "101X00", then N is 6, and the non-interleaved bit pairs are 10, 01 and 00 in the sequence. Hence, l_i is 3/6=0.5. *Rate of interleaf* decides how well the spectral information inside that input bit is interleaved. Inputs with larger rates of interleaf are easier to be merged with others, while inputs with smaller rate of interleaf are more likely to cause conflicts during merging. Selecting inputs with larger rate of interleaf also reduces loss in spectral information after merge.

$$l_i = 1 - \frac{\#\, of\ non_interleaved\ bit\ pairs\ for\ input\ i}{sequence\ length\ N} \tag{4.2}$$

Merge is the procedure of combining more than two input sequences into a single input sequence. An input pair after merging is denoted as

$$(input\ i\ of\ base\ set,\ input\ j\ of\ target\ set,\ inserted\ time\ frame) \tag{4.3}$$

When trying to combine a specified bit b_i with a don't-care bit b_j, the merged bit will simply have the value of the specified bit b_i. Conflict arises when we try to combine two specified bits with opposing logic values. In this case, the value for the bit in the merged sequence will be a don't care. Input sequences to be merged are put into either base set or target set. Firstly, sequences

in the same set are concatenated, then each sequence from the target set is to be inserted into the sequence for the base set.

Finally, we define **unbounded PI** (*UPI*) of a cluster after merging to be

$$UPI = \sum_{i=0}^{n} \#PI(T_i) - \sum_{j=0}^{m} \#PI(T_j) \quad (4.4)$$

where T_i are cores in base set and T_j are cores in target set. *UPI* indicates the number of possible configurations for merging. A merge with a larger UPI is a loose merge, while a merge with a smaller UPI is a tight merge. A loose merge has more alternations in configuring the input mappings between base set and target set, which means more choices for setting the switching network.

4.3.3 Algorithm for Interleaved State Relaxation

To merge the spectra of two sets of cores, *interleaved state relaxation* is performed for relaxing the compacted sequence of each core, in which the *rate of interleaf* for each input bit is to be **maximized**. Suppose we have the following relaxed sequences for two cores, and the two sequences are merged by inserting sequence A into sequence B at time frame 1 (insertion place), under a *direct input mapping*. Here *direct input mapping* means no switching is allowed. If the width of sequence A (W_A) is larger than the width of sequence B (W_B), input bit $i+m$ in sequence A is combined with input bit i in sequence B, where m is a constant non-negative integer called *mapping offset*, and $0 \leq m \leq (W_A - W_B)$. Each m corresponds to a configuration of *direct input mapping*.

Time Frame	Seq A	Seq B	Merged Seq
0	110110	1110	1110XX
1	XXXXXX	XXXX	110110
2	001100	0101	0101XX
3	XXXXXX	XXXX	001100
4	111101	0001	0001XX
5		XXXX	111101

In the above example, *rate of interleaf* for each input bit of each core is 1, which is the best case. If two sequences can be ideally interleaved, then no conflict will arise during merging. In reality, it may not be possible to make each input bit to be ideally interleaved, depending on the original sequence and the structure of the CUT. In our technique, if there is no conflict during merging, then we preserve the exact spectrum of each core and there would be no information loss. On the other hand, if conflicts arise during merging, then the number of conflicts decides the amount of spectral information loss. *Interleaved state relaxation* can help reduce the number of conflicts and leads to a better *spectral compression*.

Interleaved state relaxation is performed by dynamically adjusting the level of the PIs in each time frame. In the original state relaxation in [6], the priority of relaxation is determined by the computation of support sets – nodes at lower levels in the circuit are preferred for getting a **minimal** support set. We can perform interleaved state relaxation in time-frame $N-1$ by putting those PIs whose values have been specified in time frame N into restricted PI set. PIs in the restricted PI set are set to be the highest level, and those nodes who have restricted PIs in their fan-in core are also considered with less preference. Procedure COMPUTE_SUPPORT_SET_INTERLEAVE is described below in Figure 4.6, where L is a list of gates (containing only PIs and FFs) with known logic values from previous time frame.

Shown below is another example on benchmark circuit b08rst, illustrating the difference between performing interleaved and non-interleaved relaxation on improving the *rate of interleaf* on the input sequence. All bits except for input bit #1 and #10 are ideally interleaved by non-interleaved state relaxation. The significant difference shows up after performing interleaved state relaxation on input bit #1. With non-interleaved state relaxation, we obtain a *freedom* of 0.23 and *rate of interleaf* of 0.46, and the relaxed sequence “X1X10101011X0” for input bit #1 contains 7 uninterleaved bit pairs between time frame 3 and 10. While with interleaved state relaxation, we obtain an improved *freedom* of 0.46 and *rate of interleaf* of 0.92, and the interleaved relaxed sequence “X1X1X1X1X11X0” reduces the uninterleaved bit pairs from 7 to 1. The same improvement can

```
Procedure COMPUTE_SUPPORT_SET_INTERLEAVE( L, I)
  I = restricted PI set from last time frame
  support set S = L;
  while (∃ unsupported gates in L)
    g = unsupported gate in L with maximal level;
    ss = minimal support nodes for g by picking gates
    whose fan-in cones don't intersect with I first;
    add support nodes ss to S;
    for all unsupported gates i in ss
      add i to L;
    mark g supported;
  I = ∅;
  set all PI levels in I back to 0;
  Update I by adding all the specified PIs in this time frame;
  Set the level of gates in I to be the maximal level;
  return support set S;
```

Figure 4.6. Algorithm for Interleaved State Relaxation

be found on input bit #5. The only existing non-interleaved bit pair after non-interleaved state relaxation for input #5 at time frame 5 and 6 is further interleaved by relaxing input bit #5 in time frame 5 to don't care via interleaved state relaxation. For all inputs except input #10, we obtained very good *rate of interleaf* and very high *freedom*. Due to the low *freedom* of 0 and very low *rate of interleaf* of 0, input bit #10 can be a problem in merging. Nevertheless, the reason for bit #10 of b08rst is likely to be the functional reset signal of the CUT, which are not able to be relaxed or interleaved. Further discussion will be given on how to deal with reset signals by DFT insertion in later sections.

Time Frame	Interleaved	Non-Interleaved
0	XXXXXXXXX0	XXXXXXXXX0
1	1XXXXXXXX1	1XXXXXXXX1
2	X100010101	X100010101
3	1XXXXXXXX1	1XXXXXXXX1
4	XXXXXXXXX1	0XXXXXXXX1
5	1XXXXXXXX1	1XXX0XXXX1
6	XXXX0XXXX1	0XXX0XXXX1
7	1XXXXXXXX1	1XXXXXXXX1
8	XXXXXXXXX1	0XXXXXXXX1
9	1XXXXXXXX1	1XXXXXXXX1
10	1XXXXXXXX1	1XXXXXXXX1
11	XXXXXXXXX1	XXXXXXXXX1
12	0XXXXXXXX1	0XXXXXXXX1

From this example, we can see more than 90% of the bits are don't cares, and they exist in the relaxed sequence of a core. The embedded redundancy (don't-cares) provides enough room for us to merge the interleaved relaxed input sequences from different cores together, with very few or non conflicts.

4.3.4 Delay Network

Even with an ideal interleaved input sequence, (that is, every input bit has a *rate of interleaf* of 1), conflicts may still arise during merging. Take the sequences below as an example.

```
Time                    Merged Sequence
Frame Seq A   Seq B (n=0,m=0,max_delay=1)
  0   1X0X1X  0011          10011X
  1   X1X1X0  XXXX          0111X0
  2   0X0X1X  1010          00001X
  3   X1X1X0  XXXX          1111X0
  4   0X1X0X                0X1X0X
```

There are a total of 6 ways of inserting sequence B into sequence A under the condition of direct input mapping. In Table 4.3, we show the number of conflicts caused by merging, where n represents the insert position (time frame), and m is the *mapping offset*.

Table 4.3. Conflicts after Merging for Different Direct Input Mappings

	m		
n	0	1	2
0	4	1	1
1	3	1	3

This situation arises because there is a phase shift for don't cares between different input bits after *interleaved state relaxation* for sequence A. Bits 2, 4, 6 are relaxed to be don't cares in the even-numbered time-frames, while bits 1, 3, 5 are relaxed to be don't cares in odd-numbered time-frames. To solve this problem, the individual bit streams of the sequences to be superimposed onto the base set may be aligned differently along the time axis, so that less conflicts will arise during merging. By doing so, the input streams in the regenerated test patterns for some cores may be misaligned. The delay network records the amount of delay (alignment adjustment) for each input

of each inserted core when merging is performed, and is used to resynchronize the inputs of the inserted core during test application. The maximal delay allowed is directly related to hardware overhead for constructing the delay network.

Let us return to our former example. By allowing a maximal delay of 1 unit, we can easily merge the two sequence without causing any conflicts. Take the direct input mapping with insert position $n = 0$ and *mapping offset* $m = 0$ in our case. Input pairs (A1, B1, 1), (A2, B2, 0), (A3, B3, 1), and (A4, B4, 0) are formed. We are able to merge the above two sequences without any conflict and the merged sequence is shown in the rightmost column above.

4.3.5 Switching Network

In the previous subsection, we restrict the merging to be a *direct input mapping* without allowing any switch network. But what if two inputs can not share the same spectral information, such as reset signals that cannot be relaxed or interleaved without conflict logic values? Below is an example.

```
                       Merged Seq
Time                  after Switch
Frame Seq A  Seq B (n=0,m=2,max_delay=0)
  0   X00010  XXX1         100010
  1   XXXXX0  0111         1X0110
  2   X11000  XXX1         111000
  3   XXXXX0  1011         1X1010
  4   X01010               X01010
```

If we use direct input mapping with $n = 0$ and $m = 2$, then input #6 from sequence A and input #4 from sequence B form a pair. This input pair will cause conflict in *every* time-frame during merging. None of these options – changing insert position n, *mapping offset* m or increasing maximal amount of delay, can make the merging conflict free.

A better way for solving this problem is to switch the reset signal to another input with a larger freedom. Picking those input pairs whose conflicts are above a cut-off value, we separate those input pairs and re-route one of the inputs to *unbounded PI*, which still has a large freedom. After switching, the conflict during merging for every input pair is small enough, which guarantees that inputs in a pair can share the same spectral information.

In the example above, if we break the pair (A6, B4, 0), make a new input pair (A1, B4, 0) and make A6 stay by itself, we got the merged sequence with no conflict at the rightmost column. These switchings are actually the connections between different delay units in our delay network, and they form a switching network.

4.3.6 Algorithm for Spectral Compression

Shown below in Figure 4.7 is the algorithm for performing *spectral compression*, taking both delay and switching into consideration. After the procedure, we can obtain several *compressed spectrum* for the cluster under test under different configurations – different mapping, different delay network, and switching network. The same amount of test patterns are generated, using different *compressed spectrum* P_k, and the one with the highest fault coverage for the whole cluster is selected to be our final spectrum. In the procedure, the threshold value can be adjusted according to the number of *unbounded PIs* of the base set, and we generally pick it to be 0.8 times the maximal number of conflicts on a pair of inputs.

```
for each core i inside the cluster under test
    compute its characteristic fault set Ci;
    Ti = compacted test sequence for Ci;
    Si = relaxed Ti in an interleaved manner;
D = the maximal delay allowed;
put Ti of base set together to form S;
cut-off = 0.8;
for each configuration k
    G_k = S;
    for each core i in target set
        merge Si with G_k under direct input mapping;
        record conflicts for each input pair;
    Sort the conflicts in descending order;
    threshold = maximal number of conflicts * cut-off;
    for input pair whose conflict value is above threshold
        try to switch;
    perform spectral analysis on G_k;
    resulted P_k is the compressed spectrum for config k;
```

Figure 4.7. Algorithm for Performing Spectral Compression

4.4 Experimental Results

4.4.1 Experimental Results on ISCAS89 & ITC99 Benchmark Circuits

We conducted experiments of our approach on both ISCAS89 [47] and ITC99 [48] benchmark circuits. We allowed for a maximum of 70,000 vectors, generated from the spectral information extracted for the characteristic faults after our proposed *interleaved state relaxation*. We first establish that spectral method is very effective for achieving high coverages with small test sets. We test each core individually by its own spectrum without any *spectral compression* [6]. Table 4.4 compares the faults detected by spectral method with other techniques, including STRATEGATE [49], HITEC [58], PROTEST [59] and ideal weighted random BIST. For all six random resistant benchmark circuits, the spectral method achieves very high fault coverages, considering only 1 fault is used for spectrum extraction. The size of the spectrum in terms of bits is reported on the last row in Table 4.5, that is, the product of the number of PIs and the length of Hadamard coefficient set.

Table 4.4. Fault Detection of Different Techniques

Circuit	s1423	s1494	s5378	b08	b11	b12
Ideal Wgt Ran	1319	1451	3127	461	937	663
HITEC	723	1453	3231	-	-	-
STRATEGATE	1414	1453	3639	463	1003	1488
PROPTEST	**1416**	1453	**3643**	-	1004	1470
Spectral	1413	**1453**	3621	**463**	**1004**	**1663**

Only **one** characteristic fault is used for Spectral method

Next, we construct clusters by randomly grouping several cores. With *unbounded PIs* available, there may be several configurations for input sharing among cores. *Spectral compression* is performed on each configuration for the cluster, following the algorithm stated previously, inside which the delay network and the switching network are further configured. The *compressed spec-*

Table 4.5. Storage in Bit for Testing a Single Core by Spectral BIST

Circuit	s1423	s1494	s5378	b08	b11	b12
Number of PIs	17	8	35	10	8	6
Length of Coeff Set	25	12	31	23	25	161
Storage Size	425	96	1085	230	200	966

Only **one** characteristic fault is used for Spectral method

trum for a cluster under each configuration is then used to generate the same amount of test patterns, 70,000, as before. We want to point out here that once the cluster has been formed, that is, we know which cores are contained inside the cluster, the speedup and storage reduction via testing that cluster in parallel using spectrum-based BIST are fixed. The only difference between different configurations is the test quality, that is, the fault coverage of each core inside the cluster. We fault simulate the entire cluster, using the generated test patterns, and fault coverage is computed for each core inside that cluster under each configuration. The configuration with the highest fault coverage for the whole cluster is finally chosen and reported in Table 4.6.

In Table 4.6, the column *Cluster Size* reports the number of cores inside a cluster; the following six columns report the fault coverage of the cores in a given cluster configuration, and a “/” is used to indicate that core is not present in the cluster. The columns *Speedup* and *Storage Reduction* give the speedup in test application time and percentage of storage reduction by testing the cores in the cluster simultaneously, compared with serially testing the cores. The column *Speedup II* reports the speedup gained when the internal test bus is wide enough to apply each test pattern in a clock cycle, while the column *Speedup I* reports the speedup by assuming the width of the internal test bus is only 1 bit. Thus, *Speedup I* and *Speedup II* are the lower bound and upper bound in speedup that can be gained via testing the cores in the proposed parallel fashion. The *Unbounded PI* column shows the number of unbounded PIs of the cluster, which indirectly reflects the amount of merging possible. The final column *Switch* gives the number of input pairs that has been switched in the final configuration.

Table 4.6. Comparison of Fault Detection, Test Application Time and Storage Requirement By Testing Cluster as a Whole

Cluster	Size	Cores in the Cluster						Speedup	Speedup	Storage	UPI	Switch
		s1423	s1494	s5378	b08	b11	b12	I	II	Red		Red
cluster 1	2	1411	/	3616	/	/	/	1.48	2.0	0.28	18	2
cluster 2	2	/	1452	**3630**	/	/	/	1.23	2.0	0.08	27	0
cluster 3	2	/	/	**3629**	463	/	/	1.28	2.0	0.18	25	0
cluster 4	2	/	/	**3624**	/	1004	/	1.23	2.0	0.16	27	1
cluster 5	2	1413	1452	/	/	/	/	1.47	2.0	0.18	9	1
cluster 6	2	**1414**	/	/	/	1004	/	1.47	2.0	0.32	9	1
cluster 7	2	1412	/	/	463	/	/	1.59	2.0	0.35	7	1
cluster 8	2	/	1451	/	463	/	/	1.80	2.0	0.29	2	1
cluster 9	2	/	/	/	463	1004	/	1.80	2.0	0.42	2	1
cluster 10	2	/	1445	/	/	995	/	2.00	2.0	0.33	0	0
cluster 11	2	1362	/	/	/	/	1620	1.12	1.7	0.10	11	1
cluster 12	2	/	1414	/	/	/	1478	1.46	1.7	0.07	2	1
cluster 13	2	/	/	3595	/	/	**1673**	1.02	1.7	0.09	29	1
cluster 14	2	/	/	/	463	/	**1666**	1.33	1.7	0.11	4	0
cluster 15	2	/	/	/	/	1004	1482	1.46	1.7	0.13	2	2
cluster 16	3	1386	/	3601	/	1004	/	1.71	3.0	0.36	10	1
cluster 17	3	1409	/	3606	441	/	/	1.77	3.0	0.37	8	2
cluster 18	3	1401	1447	/	/	1004	/	1.94	3.0	0.41	1	2
cluster 19	3	1404	1453	3620	/	/	/	1.71	3.0	0.32	10	1
cluster 20	4	1375	1443	3581	/	1004	/	1.94	4.0	0.40	2	1

Maximal delay allowed is **0**, and cut-off value is set to be 0.8

The first 15 clusters in Table 4.6 are all of size 2. Compared with testing each core individually (shown in Table 4.4), there is almost no fault coverage loss (within **0.2%**) for the first nine clusters by using the *compressed spectrum* to do parallel testing. For some cores, better fault coverages can be achieved when tested inside a cluster, such as core s5378 in cluster 2, 3, 4 and core s1423 in cluster 6. One possible explanation for this is that the spectrum of the two single cores inside the cluster enhances each other, so the *compressed spectrum* can perform better in testing each single core. The slight loss of fault coverage arises when a cluster has a small number of *unbounded PI*, which leads to a tight merging with less choice of configurations. For example, in cluster #10, the

number of *unbounded PI* is 0. Cores s1494 and b11 in this cluster lose 8 and 9 faults respectively, and that is the place where the largest loss appears for the first 10 clusters. Although tight merging with small number of *unbounded PIs* can lead to some loss of fault coverage, it can achieve larger storage reduction and higher speedup when there is a limitation for the width of the internal test bus. For the first 10 clusters (of 2 cores), significant reduction in test application time is achieved. The difference between *Speedup I* and *Speedup II* is proportional to the percentage of *unbounded PIs*.

Clusters 11-15 all have core b12 in common. The number of test patterns generated for these 5 clusters using parallel testing is 84,000. That is why the ideal speedup Speedup II is only 1.7. Notice that there is a larger loss of fault coverage for some of them, compared with the first 10 clusters. The loss of fault coverage comes up not only because tight merging but also because of the imbalance among lengths of relaxed input sequences for the individual cores inside the cluster. According to the last row of Table 4.4, the length of the spectrum for b12 is 966 / 6 PIs = 161, which is significantly longer than some other cores, such as for s5378 (1085 / 35 = 12). It is very likely that the cores with a much smaller length cannot be tested efficiently by the *compressed spectrum* because its spectral information is only represented in a small portion of the *compressed spectrum*. Unbalanced spectral lengths among cores inside a cluster also make the storage reduction less significant. If unbalanced sequences are not avoidable in forming clusters, one possible remedy is to duplicate smaller spectral sequences multiple times along the time axis, which allows the spectral information of that core to cover more time frames in the final *compressed spectrum*.

Clusters 16 to 20 are those whose sizes exceed 2. With increased sizes of a cluster, the ideal speedup Speedup II and storage reduction both go up. Take cluster #18 as an example. It tries to test cores s1423, s1494, and b11 simultaneously. Because only one *unbounded PI* is available, it is a tight merging. We obtain 1401, 1447 and 1004 faults detected for these three cores respectively, which are 12, 6, and 0 fewer faults detected compared to single-core testing. It is clear that by using parallel testing of cores in the cluster, the ideal speedup is 3. And if the width of the test bus

is set to 1, which is an extreme case, then it will take 17+8+8 = 33 cycles to do single testing, and take 17 cycles to do parallel testing, leading to a speedup of 33/17 = 1.94. To calculate the storage reduction, we will need to use the last row on Table 4.4. It originally needs 425+96+200 = 721 bits for storing the spectrum of each core on the chip, while the size of the *compressed spectrum* equals the size of the spectrum in the base set (core s1423 in this case), which is 425. Storage reduction is (721-425)/721 = 0.41.

Finally, there is no hardware overhead for the delay network because we set the maximal amount of delay to be 0. And for the clusters tested, the maximal number of switched input pairs reported in the last column of Table 4.6 in its final configuration is **2**, indicating very little routing area overhead.

4.4.2 Quantitative Analysis on Speedup & Storage Reduction

We provide discussion below on how to select cores to form a cluster so that higher storage reduction and speedup can be achieved. For a simpler discussion, assume there is only one core C_{base} in the base set, whose spectrum S_{base} is of W_{base} bit width, and of length L_{base}. Assume also that there are N cores C_i in the target set, indexing from core 1 to core N, and their spectra S_i are represented by width W_i and length L_i $(1 \leq i \leq N)$, respectively. We next assume the following conditions in Equation Set (4.5), which says that L_{base} is the largest one among all cores in the cluster, and W_{base} is no less than the sum of W_i in the target set. This ensures that the testing time and storage requirement for testing the whole cluster in parallel to be no more than those of testing the base core C_{base}.

$$\begin{aligned} W_{base} > \sum W_i \quad & (1 \leq i \leq N) \\ \forall i, \qquad & L_i \leq L_{base} \end{aligned} \tag{4.5}$$

Number of unbounded PIs UPI can be computed using Equation (4.4), which is the same as that in Equation (4.6):

$$UPI = W_{base} - \sum W_i \quad (1 \leq i \leq N) \tag{4.6}$$

The total test time $T_{individual}$ and total storage requirement $V_{individual}$ for testing each core in the cluster individually are listed below in Equation Set (4.7), where $T_{individualUpper}$ represents the test time in clock cycles when the internal bus is wide enough to apply each pattern in one clock cycle, $T_{individualLower}$ represents the test time in clock cycles when the internal bus is **1** bit wide, and T_0 represents the number of patterns generated from the spectrum, in our case, it is 70000.

$$\begin{aligned}
Let\ i &\in \{base, 1, 2, ..., N\} \\
\forall i, \quad T_{individualUpper}(i) &= T_0 \\
\forall i, \quad T_{individualLower}(i) &= W_i * T_0 \\
\forall i, \quad V_{individual}(i) &= W_i * L_i \\
T_{individualUpper} = \sum T_{individualUpper}(i) &= (N+1) * T_0 \\
T_{individualLower} = \sum T_{individualLower}(i) &= \sum W_i * T_0 \\
V_{individual} = \sum V_{individual}(i) &= \sum W_i * L_i
\end{aligned} \tag{4.7}$$

The total test time $T_{parallel}$ and total storage requirement $V_{parallel}$ for testing all cores in the cluster in parallel are listed below in Equation Set (4.8), where $T_{parallelUpper}$ represents the test time in clock cycles when the internal bus is wide enough to apply each pattern in one clock cycle, and $T_{parallelLower}$ represents the test time in clock cycles when the internal bus is **1** bit wide.

$$\begin{aligned}
Let\ i &\in \{base, 1, 2, ..., N\} \\
T_{parallelUpper} &= 1 * T_0 \\
T_{parallelLower} &= W_{base} * T_0 \\
V_{parallel} &= W_{base} * L_{base}
\end{aligned} \tag{4.8}$$

Speedup and storage reduction via testing the cores in parallel is computed in Equation Set (4.9) below:

$$\begin{aligned}
Speed_{Upper} &= \frac{T_{individualUpper}}{T_{parallelUpper}} = N+1 \\
Speed_{Lower} &= \frac{T_{individualLower}}{T_{parallelLower}} = \frac{(W_{base} + \sum W_i)}{W_{base}} \quad (1 \le i \le N) \\
&= \frac{(W_{base} + W_{base} - UPI)}{W_{base}} = 2 - \frac{UPI}{W_{base}} \\
Reduction &= \frac{V_{individual}}{V_{parallel}} = \frac{(W_{base} * L_{base}) + \sum W_i * L_i}{W_{base} * L_{base}} \quad (1 \le i \le N) \\
&= 1 + \frac{\sum W_i * L_i}{W_{base} * L_{base}} \quad (L_i \le L_{base}) \\
&\le 1 + \frac{\sum W_i * L_{base}}{W_{base} * L_{base}} = 1 + \frac{\sum W_i}{W_{base}} \\
&= 1 + \frac{W_{base} - UPI}{W_{base}} = 1 + 1 - \frac{UPI}{W_{base}} = 2 - \frac{UPI}{W_{base}}
\end{aligned} \tag{4.9}$$

From Equation Set (4.9), we get the following points:

1. The speedup is only related to W_i, and it has nothing to do with L_i;

2. Ideal speedup (upper bound) $Speed_{Upper}$ equals the number of cores inside the cluster, $N+1$;

3. The lower bound of speedup occurs when the internal test bus is of width **1**. When UPI is 0, maximal lower bound of 2 is achieved;

4. When internal bus is wider than 1, while less than W_{base}, which means an input pattern can not be delivered in one clock cycle, the speedup is between the lower bound and the upper bound. The speedup will be proportional to the number of unbounded PIs UPI. A tight merging with a smaller value of UPI gives higher speedup, and a loose merging with a larger value of UPI gives smaller speedup. For example, in Table 4.6, cluster 5 and 6 with 9 UPI give a lower bound speedup of 1.47, while cluster 8 and 9 with 2 UPI give a lower bound speedup of 1.80;

5. The storage reduction is both related to W_i and L_i;

6. When L_i in the target set is close to L_{base}, a higher storage reduction can be achieved. If $L_i << L_{base}$, storage reduction is unnoticeable. Refer to Table 4.5 for the number of PIs and the length of the coefficient set for each individual benchmark circuit. For example, in Table 4.6, cluster 7 contains benchmark circuit b08 and b11, whose number of PIs (10 vs. 8) and length of the coefficient sets (23 vs. 25) are very close, and it gives a storage reduction of %42; while for cluster 12, the length of the coefficient set for benchmark circuit b12 is 10 times bigger than that of s1494, (161 vs. 12), and its storage reduction is only %7;

7. The maximal storage reduction is 50%, which occurs when UPI is 0 and every core in the target set is of the same spectrum length as that of the base core ($\forall 1, (1 \leq i \leq N), L_i = L_{base}$);

8. A tight merging with a smaller value of unbounded PIs UPI gives better storage reduction than a loose merging with a larger value of UPI;

Based on the above observations, to test a cluster in parallel using spectrum-based BIST for a higher speedup and a bigger storage reduction, it is better to form the cluster such that:

1. It has a small number of UPI;

2. The length of the coefficient set for each individual core is close;

4.5 Summary

We have presented an effective approach for logic BIST by employing temporal correlations on PIs, which is able to test a cluster of cores simultaneously using only one *compressed spectrum*. The advantage of our technique is that it generates a spectrum for the entire cluster of cores through much less computation. By employing *interleaved state relaxation*, the relaxed input sequences

from each core can be merged in a natural way with none or few conflicts. *Compressed spectrum* for the entire cluster is obtained by performing *spectral analysis* on the merged input sequence. We demonstrate that by parallel testing, test application time and storage requirement are greatly reduced without sacrificing fault coverage and incurring hardware overhead. In our experiments, for clusters of size 2, we report an average speedup of 1.45 by using a 1-bit internal test bus, and an average storage reduction of 0.21. For clusters of size 3, the average speedup and storage reduction are 1.78 and 0.36 respectively.

Chapter 5

Characteristic States and Cooperative Game Based Search

Starting from this chapter, the rest of the chapters focus on utilizing spatial correlations among FFs to enhance sequential ATPGs. Intelligent partitioning on the state variables can uncover important spatial correlations among FFs, such that the state traversal will favor those spatial correlations during the search. Thus, partitioning is the main thread for the next two chapters.

This chapter introduces a new cooperative traversal on the partitioned finite-state-machine (FSM) space to generate test stimuli for both sequential test generation and design validation. The new features for this logic-simulation-based automatic test pattern generation include: (1) cooperative search that exploits orthogonality is performed on two global state partition sets; (2) feedback information is dynamically extracted from characteristic states (each characteristic state examines a subset of corner cases of the design) to prune away redundant search spaces; (3) different from dynamic partitioning, interactions between two global partition sets are emphasized in the proposed approach. The proposed cooperative search is similar to a cooperative two-player game, where each partition set presents the state space for a player. Experimental results show that test vectors generated by such a cooperative search can yield both higher coverage and up to 5 times reduction

on test generation time.

5.1 Introduction

As explained in Chapter 1, generating effective vector sets for large sequential circuits remains a challenging problem in both testing and validation of digital systems. The vector sets can be generated by either deterministic or simulation-based methods. Deterministic test generators can suffer from potentially long execution times in large designs due to frequent backtracks in the enormous search spaces. Likewise, in design verification, formal methods used for property checking face the hurdle of "state explosion problem", where the complexity of the circuit model can be exponential to the number of state variables that comprise the system. On the other hand, simulation-based stimuli generation poses as a promising alternative for both testing and validation of large sequential systems. In fact, simulation is the main workhorse for design validation today.

Despite the advances made in logic-simulation-based ATPG, the coverage obtained may not be as stable as the other test generators. For instance, a number of runs of the logic-simulation-based ATPG from different random seeds may result in different coverages. Furthermore, the coverage obtained by logic-simulation-based ATPGs may still lag behind other ATPGs for some sequential circuits. Our work, thus, focuses on overcoming these two problems. Characteristic states, states that examine corner cases of the design are defined; critical information which can lead the search to reach the characteristic states is dynamically extracted along the test generation procedure. We also emphasize the interactions between two global partition sets by forming a two-player game model. These two techniques drive the search into state spaces that are harder to be explored. Experimental results show that the cooperative traversal proposed in this work outperforms other traversal methods. It detects an average of 10% more subtle bugs than the previous works [60, 14] for design validation. In addition, for most benchmark circuits tested, it achieves the highest stuck-at fault coverages compared with other simulation-based counter-part with up to 5 times reduction

on test generation time.

The rest of the chapter is organized as follows. Section 5.2 gives the overview and motivations. Section 5.3 presents detailed accounts of our approach based on characteristic states and game theory. Results are reported in Section 5.4, followed by the summary in Section 5.5.

5.2 Background and Motivations

Logic-simulation-based ATPGs have shown promise for generating effective tests for both manufacturing test and design validation. It has two sub-categories: (1) ATPG based on partitioned FSM traversal (PL-ATPG); (2) ATPG based on non-partitioned FSM traversal (NL-ATPG). The difference between PL-ATPG and NL-ATPG resides mainly in the objective function on how to direct the search in the state space. In NL-ATPG, the state variables are not partitioned and the search tries to reach as many new global states as possible. On the other hand, in PL-ATPG, the state variables are partitioned into several groups, with each group given a different weight, (e.g., partitions consisted of FFs on the control path are given higher weights than partitions consisted of FFs on the data path), and the search will only consider a global state to be new and useful by the indications from the partitioned states. In other words, a combination of previously reached partial states is not regarded as a new and useful global state.

The PL-ATPG has proved to be more efficient than NL-ATPG [12, 60], and we briefly explain the reason as follows. The underlining assumption for logic-simulation-based ATPG is that some new states are deemed more desirable than others. Because the entire global state space is of size 2^N (N is the total number of state variables in the circuit), there is a lot of noise embedded in this huge state space. This noise prevents the search from reaching useful global new state that could lead to improvement in coverage. The PL-ATPG limits the definition of a "new" state to be local around partitioned groups of state variables. Intelligent partitioning is needed to group those related state variables together. The PL-ATPG greatly reduces the size of the search space to $M * 2^{(N/M)}$, (M

is number of groups), thus it weeds out some of the embedded noise. In PL-ATPG, the size of each partition (N/M) is generally set to a small number, such as 8 or 10, to ensure that the partial state space for each partition is manageable. Shown in Equation (5.1), when N is large, the ratio between the size of the global state space and that of the partial state space is exponential.

$$\begin{aligned} \frac{|GlobalStateSpace|}{|PartialStateSpace|} &= \frac{2^N}{M * 2^{(N/M)}} \\ &\approx \frac{2^N}{M * 2^{10}} = \frac{2^{N-10}}{M} \end{aligned} \tag{5.1}$$

For a given partition set, two types of correlation among FFs are possible:

1. *intra-partition correlation*: correlations among FFs in the same partition;
2. *inter-partition correlation*: correlations among FFs across different partitions;

Shown in Figure 5.1, intra-partition correlations and inter-partition correlations are differentiated by different types of arcs. Without partitioning (shown at the top of the figure), all correlations are intra-partition correlations. By introducing state partitioning, a large percentage of intra-partition correlations become inter-partition correlations. The number of new states is greatly reduced. This is because only intra-partition correlations among FFs are considered, and inter-partition correlations are ignored for the most part. The effects it brings are two-fold: (1) it filters noise among the correlation set, and partial new state is proved to have closer relation to improving coverage; (2) it may inadvertently filter some useful correlations, so some very hard-to-detect faults may still remain undetected. To compensate for the second issue, dynamic partitioning has been proposed in [12], in which the original partition set is discarded and the FFs are repartitioned. For example, in Figure 5.1, suppose the detection of a hard fault f requires the value “10” on FFs #1 and #3. Without any state partitioning, it may be very difficult to drive the search to “10” on FF #1 and FF #3 simultaneously, as the number of global states is huge and there is no discrimination among the states. When partition set 1 is used, correlations between FF #1 and FF #3 are inter-partition

correlations, which are not part of the objective function of the search. Subsequently, this value combination will not be favored by the GA, and f may remain undetected. After repartitioning of the FFs to partition set 2, FF #1 and FF #3 now belong to the same group. The correlations between FF #1 and FF #3 are intra-partition correlations and will be favored by the search. Consequently the hard fault f has an increased chance of being detected.

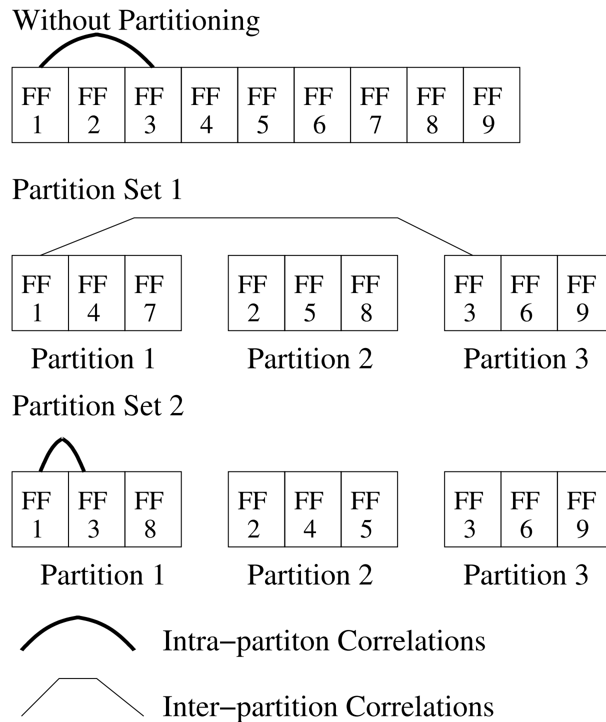


Figure 5.1. Different Types of Correlations

5.2.1 Motivation Behind Our Approach

The basic flow of the PL-ATPG we employ is shown in Figure 5.2. Random vectors are first generated. Information embedded in the set of reached states by the random vectors is then used to partition the state variables. A genetic algorithm (GA) is used to generate test vectors on the

partitioned state machine. The population for the first generation of the GA is randomly generated, which contains n sequences of length l. The GA iterates over m generations. The offsprings whose fitness values are above a threshold from generation $i-1$ are passed to generation i. After m generations, the vector sequence with the best fitness is selected and appended to the test vector set V. This process continues until the number of vectors in V reaches a predefined number. The **State Traversal Condensation** block in the figure is targeted at reaching the partial states via a shorter input sequence. It can be regarded similar to a test sequence compaction, except that no fault is considered and only logic simulation is involved. Details on state traversal condensation will be given later in the chapter.

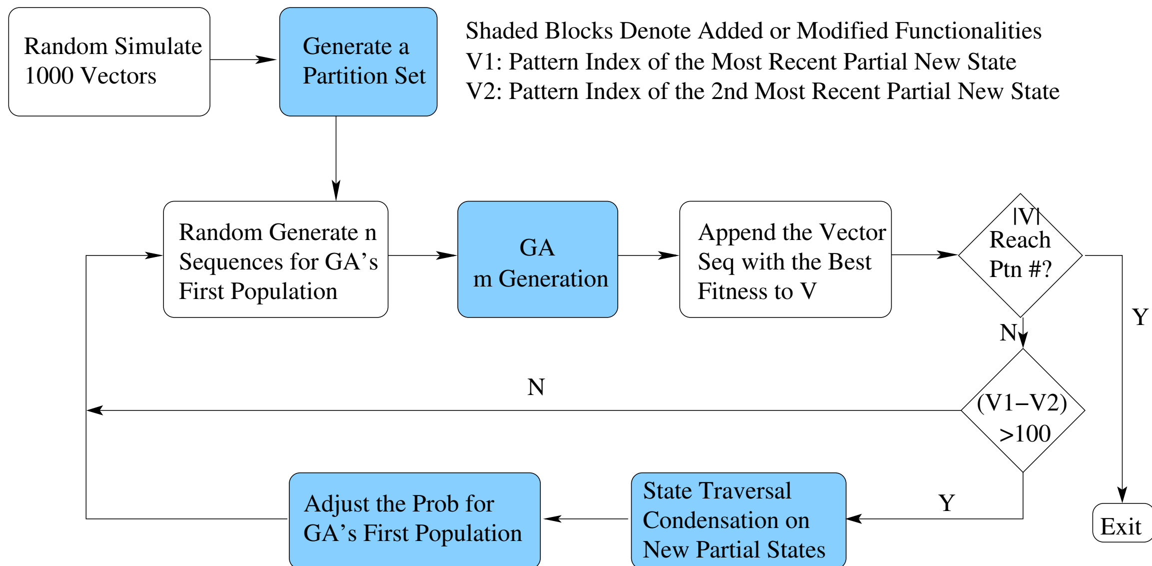


Figure 5.2. Comparison of the Framework Between Previous Work and Current Work

5.2.1.1 Explore Beneficial Interactions

Although dynamic partitioning has shown to perform better than without repartitioning, there still exist some useful correlations that it cannot catch. Figure 5.3 illustrates the reason. In the left half of the figure, the two circles represent the state spaces (only consider intra-partition correlations) that are favored by partition sets I and II respectively. Individual dots denote global states and

directed edges between two dots denote state transitions. A large portion of the state spaces of partition sets I and II overlap, indicating that much of the state space can be reached by either partition set. Nevertheless, each partition set has a unique partial state space that is more likely reachable only with its corresponding partition set. For example, states A and C are more likely to be reachable under partition set I; on the other hand, state B is more likely to be reachable under partition set II. If a hard-to-detect fault f requires the state transition $A \to B \to C$ for its detection, dynamic partitioning that switches from partition set I to partition set II still cannot detect f, because state C is back in the state space favored by partition set I. Without interactions between the two partition sets, there is little chance that the state transition sequence $A \to B \to C$ may be selected.

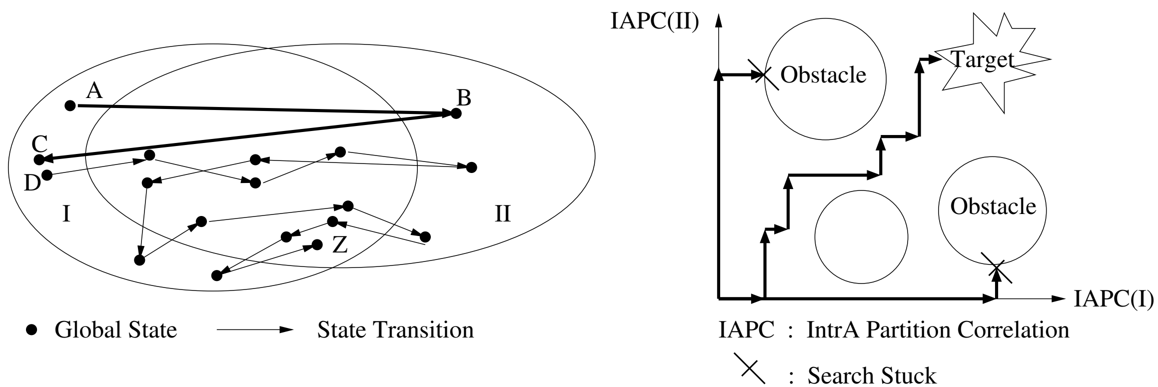


Figure 5.3. Illustration of Cooperative Search

Some harder faults may require more interactions between two partition sets for their detections, shown by the other directed path from state D to state Z in the left part of Figure 5.3. This path would require multiple alternations between the state spaces of the two different partition sets. This inspires us to investigate how multiple partition sets may be explored cooperatively, instead of focusing only on one partition set at a time. If the interactions between two partition sets are beneficial, it is similar to performing orthogonal search to approach the target, shown in the right half of Figure 5.3. Orthogonal search can be modeled as a two-player cooperative game (in a

cooperative game, the players are coordinated to attain the best result for the entire group); the interactions (cooperations) between two players provide more flexibility in avoiding stuck at the obstacle states.

5.2.1.2 Space Pruning

Figure 5.4 explains the procedure of using closed-loop GA to prune the PI search space. Closed-loop strategies have shown to be very useful in design verification [61], where a Bayesian network is put on the feedback path. The basic idea for a closed-loop search is to extract useful information from the run of the previous iterations to direct the search/prune the search space in the current iterations. Useful feedbacks extracted from previous iterations can accelerate the search procedure or hit the search target from different directions. For the best efficiency in closed-loop strategies, the key is on how to extract useful information.

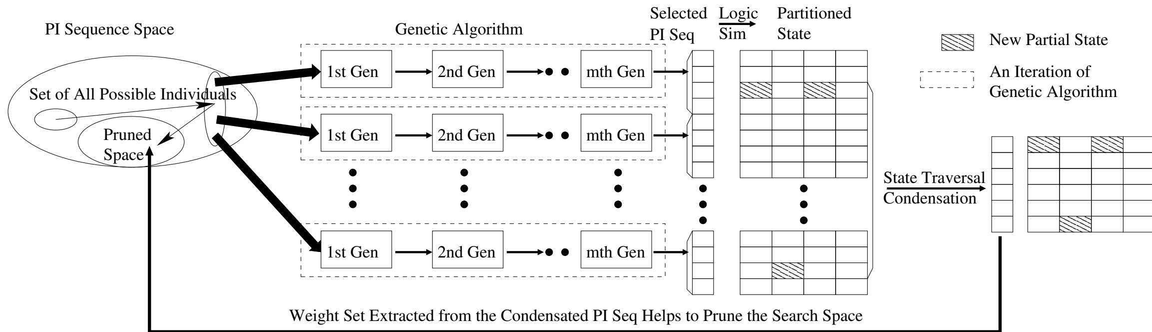


Figure 5.4. Illustration of Closed Loop Genetic Algorithm

In our application, the search space comprises of all the possible test sequences, and the target is the reaching of new partial states. Starting with the randomly generated population for the first generation, the GA goes for m generations and picks the individual with the best fitness. This is called a *GA iteration*. The individuals selected by different GA iterations will be concatenated to form the final test set V. In order to refine the evolutionary process, we introduce the concept of characteristic-state based state-traversal condensation to extract the useful information on the

feedback loop. A *characteristic state* is defined as a partially specified state which can help with reaching the corner cases of a design. In other words, it represents a cluster of hard-to-reach states in the state space. In our work, when a characteristic state is found, state traversal condensation is invoked dynamically to extract the necessary information which leads search to reach this state. The extracted information is used as knowledge that is fed back to initialize the first population in the next GA iteration. This actually prunes the PI search space, and limits the GA to search only in a much smaller but much more valuable PI space.

In this work, we emphasize on two aspects in the test generation process:

1. Investigate interactions between two partition sets by modeling the alternative search procedure as a cooperative two-player game;
2. Employ characteristic states to form the closed-loop feedback mechanism and dynamically direct the search.

Before we delve into our approach, some relevant background materials on game theory and genetic algorithm are first given.

5.2.2 Overview of Game Theory

Game theory is a distinct and interdisciplinary research originated from the study of human behavior on economic patterns. Since the work pioneered by von Neumann, Morgenstern, and Nash, "games" have become a scientific metaphor for a much wider range of interactions in which the outcomes depend on the interactive strategies of two or more players with opposed or mixed motives. Game theory addresses the series of interactions using the metaphor of a game: in the series of interactions, as in a game, the individual's decision is essentially a choice of a strategy, and the outcome of the interaction depends on the strategies chosen by each of the participants. From

recent literatures, game theory has been successfully applied to the areas such as adaptive wireless networks [62], power management of antenna networks [63], and competition-driven SAT (boolean satisfiability) solvers [64].

Games can be classified into two categories: non-cooperative games and cooperative games. In a non-cooperative game, each player maximizes its own rewards. On the other hand, in a cooperative game, the players are coordinated to attain the best result for all the players. We list some basic concepts for a cooperative game.

1. *coalition*: A group of players who commit themselves to coordinate their strategies in a cooperative game;
2. *side payment*: Bribe arrangements so that some players might be better off;
3. *allocation/imputation*: After all the bribes and side payments have cleared, the outcome that the members of the coalition get;
4. *efficient allocation*: The allocation is efficient if no one in the coalition can be made better off without making someone else worse off;
5. *cooperative solution set*: The set of all efficient allocations is called the cooperative solution set;

The famous Prisoner's Dilemma [65] is used to explain the above concepts. Two burglars, Al and Bob, have been captured and have been given a burglary charge separately by the police. Each has to choose whether or not to implicate the other. If neither man implicates the other, then both will serve separate one-year sentences. If both implicate each other, then each will go to prison for 10 years. However, if Al implicates Bob, but Bob does not implicate Al, then Al will go free, while Bob will be imprisoned for a maximum penalty of 20 years. Two actions are possible for each man in this case: implicate the other or don't implicate. The payoffs (penalties, actually) are the sentences served. We can express this compactly in a "payoff table" as shown in Table 5.1, where

each entry of $(a,b)/c$ indicates the following: the ordered pair (a,b) indicates the payoffs to Bob and Al, while the c represents the sum of the payoffs to both individuals. For example, (0, 20)/20 in the table can be explained as: payoff to Bob is 0 years; payoff to AL is 20 years; payoff to Bob and AL is 20 years.

Table 5.1. Payoff Table for the Prisoner's Dilemma

	AL	
Bob	Implicate	Not Implicate
Implicate	(10, 10)/20	(0, 20)/20
Not Implicate	(20, 0)/20	(1,1)/2

Each individual might think, to implicate the other indicates either 0 (in the case where the other does not implicate back) or 10 years (in the case where the other implicates) of prison, which is a less amount of penalty when compared to not implicating the other: 1 (in the case where the other does not implicate) or 20 (in the case where the other implicates). Independent of the other's action, to implicate the other brings some benefit over not to implicate, and thus to implicate is a better strategy for an individual. So most likely, both of them will implicate each other, leading to 10 years in prison for both of them. In this non-cooperative game, each player tries to beat the other, and this results in a total number of 20 years of prison for them as a group.

Now, if they could communicate and commit themselves to coordinated strategies by forming a cooperative game, the interactions will lead to quite different outcomes such that both of them will choose not to implicate the other and each serves only a one-year sentence. Committing to form a coalition, Bob and AL in this coalition will make the agreement not to implicate each other. This leads to the final game outcome (1, 1)/2, indicating the allocation for the coalition is 2 years, which is the efficient allocation with the best outcome; this will be the cooperative solution chosen from the cooperation solution set. Recalling that an allocation is efficient if on one in the coalition can be made better off without making someone else worse off, all allocations in Table 5.1 except for (10, 10)/20 are efficient allocations. Note that in this simple example, there is no side payment.

It turns out that each player can benefit from cooperation, and so does the coalition. From this example, we can see the important role interactions play in a cooperative game to help achieve better outcomes for the players.

5.2.3 Basic Overview of Genetic Algorithms

Genetic algorithms belong to the class of evolutionary computing. Problems are solved by an evolutionary process resulting in a best (most fit) solution (survivor).

1. *individual/population*: A candidate solution for the problem at hand;
2. *fitness*: Each individual is associated with a fitness, which measures the quality of this individual for solving the problem;
3. *selection/crossover/mutation*: Typical GA operations to reproduce a new population from the existing population;
4. *generation*: When a new population is reproduced from a previous population via GA operations.

A population of individuals evolves over a number of generations using the GA operations of selection, crossover, and mutation, and the average fitness of one generation is expected to improve over the previous generations. The evolutionary process ends when the desired target is achieved or a preset amount of resources have been exhausted.

5.3 The Proposed Approach

5.3.1 Cooperative Search Using Game Theory

We apply the theory for cooperative game to our state space traversal. The motivation shown in Figure 5.3 is further explained in Figure 5.5. If each partition set represents the space (only consider intra-partition correlations) a player can move about, the partial new state space each player can explore is a subset of the global new state space. The shaded as well as pattern-filled circles represent the state spaces which can lead to fault/bug detections, and we call them *productive state spaces*. These productive state spaces are scattered inside the global state space. It is possible that the partial state space of one partition set cannot cover all the productive state spaces. To compensate for those spaces, another partition set II is added, with an attempt to cover the productive state spaces missed by partition set I as much as possible. Shaded circles are the easy-to-reach productive state spaces, which can be reached by both partition sets. On the other hand, the pattern-filled circles are those hard-to-reach productive state spaces, which will be favored by one or none of the two players.

Based on the observation that some state spaces are only favored by one of the two players, we would like to position the two players in the same coalition and perform the search in the following way:

1. State spaces that are unique to one player has a higher priority during the search, for it is considered to contain some hard-to-reach states;
2. State spaces that are favored by both players shall not be over-counted;
3. Player I and player II alternate in their decision making. The strategy that the active player made in its run tries to maximize the *allocation* (outcome) for the entire *coalition* (player I + player II), and gives the non-active player some *bribes* (advantages) in its following run;

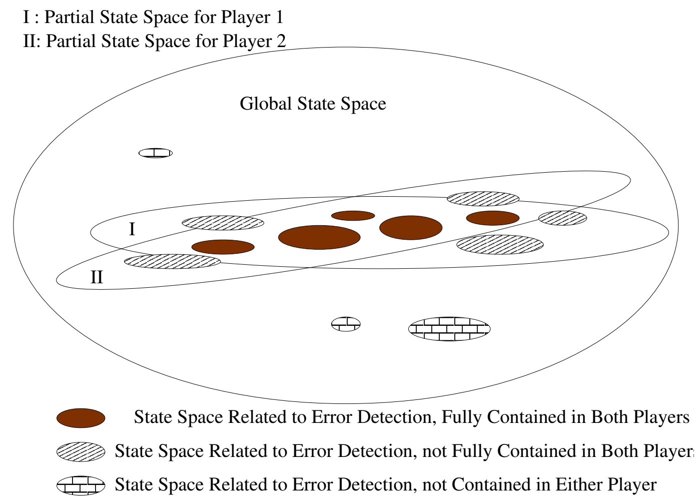


Figure 5.5. State Space Search via a Two-player Game

This strategy is different from dynamic partitioning, where the two or more existing partition sets are considered in a round-robin fashion without interacting with other partition sets. The interactions between two partition sets are emphasized in the cooperative search; this can prevent the search from being stuck in a local maximum.

5.3.1.1 Abstract Models

Players: The two partition sets make up the state spaces for the two players. Both sets are extracted from the structural information of the circuit-under-test (CUT). An observation was made in [12] that during state space traversal, FFs belonging to the control path of the CUT usually play a more important role than FFs belonging to the data path. Thus, we form two partition sets: one partition set is based on the ability of an FF to control the value of other FFs, while the other partition set is based on the ability of an FF to receive the value from other FFs. For each FF, the number of FFs in its fanin and fanout cones are computed as measurements C and D to indirectly represent whether it is more likely to be on the control path or on the data path. Then, the FFs sorted based on the descending values of measurement D are put to a list $Sort_H_iFF$, and the FFs sorted based on the

descending values of measurement C are put to a list $Sort_V_iFF$. Player I groups 10 consecutive FFs at a time from the $Sort_H_iFF$, and assigns higher weights to partitions with larger values of D, while player II groups 10 consecutive FFs at a time from the $Sort_V_iFF$, and assigns smaller weights to partitions with larger values of C.

Allocation: The two players form a coalition. The allocation for this coalition, that is, the benefits brought to the whole coalition when a player moves a step, is defined as the first two items in Equation (5.2), where F, R_F denote *Fitness* and *Resume_Fitness*, and AP, NAP denote active player and non-active player, respectively. Equation (5.2) as a whole acts as the complete fitness function.

$$F(AP) + |F(AP) - F(NAP)| + R_F(NAP) \tag{5.2}$$

In Equation (5.2), by adding the second item, the difference in fitness between two players is emphasized. This not only forces the search to prefer the new partial states that belong uniquely to one player, but also prevents the new partial states that are favored by both players from being over-counted; thus search criteria (1) and (2) are met.

The fitness for a sequence of vectors is computed similarly to that in [13], shown in Equation (5.3), where $T_i(j)$ denotes the fitness for the ith partition on the jth vector, and W_i denotes the weight assigned to the ith partition. $T_i(j)$ is 10 if the reached state s brings a new partial state to partition i; otherwise, $T_i(j)$ is $1.0/F(i,s)(F(i,s) > 0)$, where $F(i,s)$ is the number of the times partial state s has been reached in the partial state space of partition i. The sum of $T_i(j)$ along the vector sequence on all partitions forms the fitness for the current vector sequence, where l and M are the length of the test sequence and the number of partitions respectively.

$$\begin{aligned} T_i(j) &= 10 \text{ if } s \text{ is new} \\ T_i(j) &= 1.0/F(i,s) \text{ if not} \\ Fitness &= \sum_{i=1}^{M} (W_i * \sum_{j=1}^{l} T_i(j)) \end{aligned} \tag{5.3}$$

$$Resume_Fitness = \sum_{i=1}^{M} W_i * T_i(l) \quad (5.4)$$

Side Payments: Side payments allow the current active player to place the non-active player in a better resuming position, preparing for the non-active player's next run. It is defined in Equation (5.4) as *Resume_Fitness*. The active player picks a test sequence during its run, and the last vector in the sequence decides the initial state for the non-active player, thus this state is the direct interface between the two players. The active player favors the selection of the last vector such that the corresponding interface state brings many new partial states to the non-active player's partial state history table; this is the bribe that the current player gives for the other player's run. Because some states will be favored by one player over the other, if the active player can set the non-active player to start the search from a hard-to-reach state of the non-active player, it will help the next player to be able to perform a a more productive search, that is, reach some previously unexplored states.

5.3.2 Characteristic States

5.3.2.1 Definition of Characteristic States

A *characteristic state* is defined as a partially specified state which can help reach the corner cases of a design. In other words, it represents a cluster of hard-to-reach states in the state space, and the sequence that explores the characteristic state also has a high probability to explore many of the hard-to-reach states in the cluster. We give an example of characteristic states on a design with 6 state variables. If the value "11" is a hard-to-reach combination on FF #3 and FF #4, a characteristic state would be "XX11XX" and it is a representative of the state subspace that contains the following 16 hard-to-reach states: {"00**11**00", "01**11**00", "10**11**00", "11**11**00", "00**11**01", "01**11**01", "10**11**01", "11**11**01", "00**11**10", "01**11**10", "10**11**10", "11**11**10", "00**11**11", "01**11**11", "10**11**11", "11**11**11"}.

Characteristic states are obtained dynamically along the traversal on the partitioned FSM state spaces. In the early stages of the test generation, the state coverage in each partition is expected to increase at a steeper rate, indicating that new partial states are steadily obtained. These partial states are considered to be the easy-to-reach states, which do not belong to the category of characteristic states. As the search continues, the state coverage in each partition starts to level off. Any new partial state obtained during the saturating period of the state coverage is considered as a hard-to-reach case, that is, a characteristic state.

5.3.2.2 Extract Information from Characteristic States

In [6], characteristic faults are proposed to direct the pattern generation of a fault-simulation-based ATPG. As the representative of a cluster of detectable faults, a characteristic fault has the good property that a sequence that detects the characteristic fault can also detect most other faults in the cluster.

In our work, we propose state traversal condensation targeting characteristic states to extract useful information, and the procedure is invoked during the pattern generation dynamically. The key is to eliminate the unnecessary patterns from the pattern set without sacrificing characteristic states. During test generation, once a new partial state arises, the corresponding vector that brings the new partial state is recorded in $1stPtnIndex$, and the previous value in $1stPtnIndex$ is copied to $2ndPtnIndex$ to represent the second most recent pattern that can bring new partial states. The procedure *State_Traversal_Condensation* shown in Figure 5.6 is invoked whenever a test pattern has brought about a new partial state and the gap between $1stPtnIndex$ and $2ndPtnIndex$ exceeds a preset number. The new partial states corresponding to $1stPtnIndex$ and $2ndPtnIndex$ are selected as characteristic states. In our case, we choose the preset gap to 100. This is because if the size of the input pattern set fed to the procedure is very small, the “noise filtering” effect of the procedure would not be very significant. N_Length and C_Length are the sizes of input patterns before and after state traversal condensation, respectively. The ratio between $\frac{N_Length}{C_Length}$ is the condensation

```
State_Traversal_Condensation(1stPtnIndex,2ndPtnIndex)
{
    N_Length = 1ndPtnIndex - 2stPtnIndex + 1;
    Condensation_Done = 0;
    for (i=0; i<N_Length-1; i++)
        Current_State = ResumeState(2stPtnIndex);
        for (j=i; j >= 0; j–)
            Ptn = Ptn_Array[1ndPtnIndex-j];
            Current_State = LogicSimulation(Ptn);
        Condensation_Done = CheckNewState(Current_State);
        if (Condensation_Done == 1)
            C_Length = i+1;
            break;
    return(C_Length);
}
```

Figure 5.6. Algorithm for State Condensation

ratio, which varies from 1.0 to several thousand in our experiments. A big advantage related to this state traversal condensation is that it runs very fast because no fault simulation is involved.

The right half of Figure 5.4 explains the state traversal condensation procedure. PI sequences of length 4 are generated from GA iterations and are logic-simulated to obtain the reached states. States are partitioned into 4 partitions; a pattern filled block represents a new partial state, while a blank block represents a partial state that has already been reached in the state history table. State traversal condensation is then performed to ensure that the new partial states remain, as other partial states may be removed from the shortened sequence. The probability for every PI having a logic 1 is extracted from the compacted sequence, which prunes the PI search space to a much smaller content. The population for the first generation of later GA iterations will be constructed from

the pruned PI space. This procedure is repeated when the characteristic states show up, and will redirect the first population of the GA to a new state subspace, shown as the directed multi-point line in the left part of Figure 5.4.

5.4 Experimental Results

We implemented the proposed cooperative traversal in C. Experiments were conducted on ITC99 [48] benchmark circuits over a 2.8 G Pentium 4 with 1G RAM, running on Red Hat Linux. When targeting single stuck-at faults, we do not repeatedly report those benchmark circuits that have already achieved the highest test coverage in our previous work [13]. In this chapter, we focus on analyzing the effectiveness of our two-player approach, along with discussion on those circuits whose coverages are not the highest via PL-ATPG, compared with those in the literature. For each benchmark circuit, a maximum number of 80,000 vectors are generated.

To demonstrate the effectiveness of the proposed techniques, experimental results are reported and compared in Table 5.2 under 4 different options by disabling some or none of the proposed techniques. In Table 5.2, the column *CKT* reports the name of the circuit. The columns under *Option I* have both characteristic states based condensation and cooperative search disabled; the columns under *Option II* report the results with condensation disabled but cooperative search enabled; the columns under *Option III* have condensation enabled but cooperative orthogonal search disabled; finally, the columns under *Option IV* report the results where both condensation and cooperative search are enabled. For each circuit, under each option, the program runs for 20 times from different random seeds, and each run generates 80,000 vectors targeting at single stuck-at faults. The parameters of GA are set the same for all runs: $m = 3$, $n = 4$, and $l = 10$. The maximum coverage among the 20 runs, and the standard deviation in the coverage over the 20 runs are reported under *Max Coverage* and *Coverage Deviation %*, respectively.

From Table 5.2, it is observed that traversal in the state space performs the best when both charac-

Table 5.2. Comparison of Coverage and Coverage Deviation with/without Condensation & with One/Two Players

CKT	Option I Without Condensation One Player		Option II Without Condensation Two Players		Option III With Condensation One Player		Option IV With Condensation Two Players	
	Max Coverage	Coverage Deviation %	Max Coverage	Coverage Deviation %	Max Coverage	Coverage Deviation %	Max Coverage	Coverage Deviation %
b04	**1168**	**0**	**1168**	**0**	**1168**	**0**	**1168**	**0**
b05	**886**	0.24	**886**	0.56	**886**	**0**	**886**	**0**
b06	**186**	**0**	**186**	**0**	**186**	**0**	**186**	**0**
b07	**837**	**0**	**837**	**0**	**837**	**0**	**837**	**0**
b08	**463**	1.78	461	1.91	**463**	**0**	**463**	**0**
b09	333	5.53	338	5.72	**341**	**0**	**341**	**0**
b10	453	1.86	456	1.80	**467**	0.93	**467**	**0.83**
b11	996	0.63	**1003**	0.71	995	0.88	**1003**	**0.68**
b12	1459	12.0	1488	8.18	1606	1.62	**1612**	**1.58**
b13	792	1.09	**797**	0.80	793	1.04	795	**0.61**
b14	7864	1.08	7742	1.18	8202	**0.69**	**8234**	**0.69**
b15	8248	5.32	9295	4.84	**10850**	4.17	10726	**3.31**
b22	25693	1.45	28564	3.24	28962	**0.94**	**29287**	1.09
AVG	3798.3	2.38	+3.9	2.05	+12.9	0.77	**+13.4**	**0.70**

Deviation is Calculated over 20 Iterations

teristic states based condensation and cooperative search are enabled. For the smaller circuits, all four options were able to reach the maximum coverage. However, for the larger circuits, Option IV was able to perform much better. Among the 13 circuits tested, only b13 and b15 did not report the highest max coverage, and only b22 did not report the smallest coverage deviation under Option IV. Option I performs the worst where both condensation and cooperative search were disabled. Compared with the results when both techniques were turned off, we can achieve an average of 13.4% increase on the maximum test coverage and an average of 71.4% reduction on the coverage deviation. This demonstrated that the two techniques we proposed in previous sections were very efficient, which could bring significant improvement.

Results for the four large ITC benchmark circuits, b13, b17, b19 and b22, which did not achieve the highest test coverages in [13], are listed in Table 5.3. Test coverage, test set size, and test generation time are reported among several simulation-based ATPGs, such as PROPTEST [59], STRATEGATE [66] and [13]. For all four circuits, the cooperative traversal and characteristic states based ATPG achieves the highest coverage in the shortest time. Take b22 as an example, our method was able to detect 30997 faults, which is 713 more than PROPTEST, 209 more than STRATEGATE, and 296 more than [13]. In addition, the pattern generation time is only 1/4 of [13]. Except for b19, the test generation time has been significantly reduced via our new approach. The reason that we did not achieve dramatic speed up for b19 is because among the 286733 faults in b19, only a small portion (5%) of them are detectable. Fault dropping cannot be performed on the undetected faults, which makes fault simulation very slow. In fact, 83% of the test generation time in our approach was spent on the one-time fault-simulation of the test set at the end of the pattern generation in order to report the fault coverage.

To demonstrate that the interactions introduced in cooperative search play a critical role in examining the corner case of the design and finding subtle bugs, a case study on state coverages under different search strategies is reported for b13, shown in Table 5.4. Before the search started, two partition sets are formed in the way described in Section 3.1.1. In this table, *P1+P2* denotes

Table 5.3. Comparison of Fault Coverage on Large Benchmark Circuits

CKT	Fault-Simulation-Based ATPGs				Logic-Simulation-Based ATPGs					
	PROPTEST [59]		STRATEGATE[66]		[13]			Ours		
	Cov	Len	Cov	Len	Cov	Len	Time	Cov	Len	Time
b13	702	382	729	5700	596	6334	300s	**795**	9005	**68s**
b17	18740	20415	19095	48629	18982	52326	8.52h	**19508**	25140	**2.33h**
b19	13112	1701	12944	3007	12088	2808	**60.1h**	**13464**	5020	120.2h
b22	30284	33067	30788	22059	30701	38312	5.3h	**30997**	46064	**1.17h**

performing a cooperative search; *P1* and *P2* denote performing the search with the objective that maximize the state coverages of a single partition set *P1* and *P2*, respectively; *P1* → *P2* denotes exploration of partition set I followed by partition set II (i.e., exploration of partition set II begins only after the state coverages of partition set I has been saturated); and *P2* → *P1* can be explained similarly. The circuit b13 contains 53 state variables. By setting the partition size to 10, each partition set contains 6 partitions, which are listed as *Part I* to *Part VI* in the table to report the number of partial states reached in the partition.

Table 5.4. Comparison of State Coverage on Each Partition Under Different Traversal Methods for b13

Search Strategy	Fault Coverage	Partition Set I						Partition Set II					
		Part I	Part II	Part III	Part IV	Part V	Part VI	Part I	Part II	Part III	Part IV	Part V	Part VI
P1+P2	**795**	**216**	**104**	**34**	183	**492**	**8**	**331**	**128**	**78**	**48**	199	**8**
P1	769	**216**	**104**	22	162	447	**8**	**331**	**128**	60	46	178	**8**
P2	775	**216**	**104**	22	**207**	452	**8**	**331**	**128**	61	46	**226**	**8**
P1 → P2	774	**216**	95	22	179	474	**8**	**331**	**128**	58	**48**	202	**8**
P2 → P1	774	**216**	86	22	163	443	**8**	**331**	**128**	48	40	176	**8**

From Table 5.4, it is observed that the cooperative search achieves (*P1+P2*), the highest fault coverage of 795. In addition, with the cooperative search, among the 6 partitions in partition set I and the 6 partitions in partition set II, maximum state coverages were achieved for 10 of them,

namely every partition except partition IV in set I and partition V in set II. In other search strategies, the state coverage were consistently lower, as indicated from this table. This demonstrates that the cooperation between two players can improve the partial state coverages. Furthermore, the benefits brought by the two-player model are unique to the cooperative game, which cannot be replaced by dynamic partitioning, shown in the *P1* → *P2* and *P2* → *P1* rows in the table.

We also performed experiments for design validation under the gate substitution error model, where a gate is changed to a different gate type with the same number of inputs. We first built the error-list of all possible gate substitution errors in the CUT. Then, a maximum number of 20,000 vectors are generated using the proposed test generator. The vector set is simulated both on the good circuit and on the buggy circuit, each of which a single gate substitution error is injected. The results of our new approach are compared with random vector generation, [60], and [14], reported in Table 5.5. For each circuit, the error coverage is reported. It is seen that our new approach consistently outperforms other methods on the error coverage. Among the 7 circuits tested, only the coverage of b12 is slight lower (2% lower) than that obtained in [14]. In general, the coverages of other circuits have an average of 10% increase over the other methods. Because only logic-simulation is involved, the test generation is very fast, all of which is within 15 minutes. This again demonstrates that this error-model-independent, logic-simulation-based test generator is both efficient and effective for testing as well as validation.

5.5 Summary

In this chapter, we have presented a new cooperative game based traversal on the partitioned FSM space to explore the spatial correlations among FFs in helping generate efficient test stimuli for both sequential test generation and design validation. Via partitioning, some spatial correlations among FFs that are deemed to be useful to help reach corner cases of the design will be favored by the search. Inspired by game theory, a cooperative search exploits orthogonality on two global partition

Table 5.5. Coverage of Gate Substitution Error

CKT	Random Vector	[60]	[14]	Ours
b04	0.47	0.76	0.80	**0.92**
b07	0.56	0.76	0.81	**0.85**
b09	0.35	0.74	0.75	**0.90**
b12	0.28	0.58	**0.60**	0.58
b15	0.34	0.76	0.77	**0.89**
b22	0.37	0.73	0.74	**0.82**
AVE	0.39	0.73	0.74	**0.83**

sets to enhance the test generation. Furthermore, feedback information is dynamically extracted from characteristic states to prune away redundant search spaces and accelerate the error detection procedure. Experimental results show that the test vectors generated by the proposed method can achieve an average of 10% higher coverage for design validation. For some benchmark circuits, our approach achieves the highest single stuck-at fault coverage with up to five-fold reduction on test generation time.

Chapter 6

A Hybrid Property Checking Engine

The work in this chapter is similar to that in Chapter 5 in the way that important spatial correlations among FFs are emphasized via intelligent partitioning on the state variables of the CUT. Unlike in Chapter 5, where spatial correlations are extracted via structure-based partitioning, a property-dependent partitioning method is proposed in this chapter to uncover the important spatial correlations among FFs for verifying a specific property.

We present a new hybrid verification technique to seamlessly combine formal-method-based partitioning with logic-simulation to verify safety properties of large sequential hardware designs. Intelligent partitions are first formed on the state variables by quickly extracting logic correlations via a fast approximate preimage computation for each target state variable. Then, simulation is performed to explore the partitioned state space. In our approach, the target space (consisting only of the state variables related to the target property) is differentiated from the search space, and Boolean variables defined as "lighthouses" are injected on the partitioned target space to indicate the satisfiability of some portions of the property. Together with lighthouses, spot optimization is employed within a Genetic Algorithm (GA) framework to adjust the fitness evaluation dynamically during the search, thus avoiding local minima and making the search in a closed-loop, self-adjusting fashion. Experimental results on large benchmark circuits have shown that the proposed

hybrid verifier achieves equal or better results compared with the existing formal verification and simulation-based techniques.

6.1 Introduction

In recent years, model checking has gained popularity as a platform of verification, where specifications of the system are expressed as temporal properties to represent the expected system behavior, and a model checker is used to determine whether the design satisfies the given set of properties. The current model checking methods can be classified into four categories: (1) explicit methods, (2) symbolic methods, (3) simulation-based methods, and (4) ATPG-based (Automatic Test Pattern Generation) methods. Explicit methods [67] build the finite state model of the design and explicitly enumerate the states. Symbolic methods generally use ROBDDs (Reduced Order Binary Decision Diagrams) to compactly store and manipulate the set of reachable states and the state transition functions. Thus, it is able to handle larger designs than explicit methods. Although applicable on small to medium size designs, symbolic methods still face the well-known "state explosion" problem on large real designs. Simulation-based methods have been the traditional way for design validation because of its simplicity and scalability. However, it is not feasible to exhaustively simulate a design in almost all practical applications. As designs get larger, (which often imply larger state spaces), the probability that the generated tests can uncover subtle bugs or reach corner cases is significantly reduced. Thus, the effectiveness of simulation-based method largely depends on the quality of simulation vectors. To efficiently generate high quality simulation vectors, the underlying coverage metric acting as the feedback mechanism plays a critical role. ATPG-based methods [68, 69] map every property to a single stuck-at fault by constructing a property-dependent monitor circuit, and the verification task becomes how to detect those target faults. Any sequential ATPG tool can play the role as a verifier.

Formal and simulation-based methods each has its own strengths. Formal methods explore the

state space of the circuit-under-verification and store the extracted functional (logic) information in ROBDDs at the same time, but are limited by their capacities to handle large designs. In the worst case, symbolic methods will exhaust the complete search space before they find the solution. On the other hand, simulation is the main scalable workhorse for validation today. Unlike symbolic methods, simulation-based methods perform a "random walk" in the state space, hoping to hit the "target" in a limited number of steps. Each step includes evaluating a number of possible moves; each move is associated with some *fitness function*, which measures how closely the current move can bring the state of the circuit to the target state. The move with the maximal fitness value is selected as the next step. This procedure goes on until the target is hit or some resources run out. The efficiency of the simulation-based method highly depends on the metric of goodness of the generated stimuli, which guides the search in the state space. The strength and weakness of symbolic methods and simulation-based methods seem to be complementary, which inspire us to seek a way to combine them for a more robust verifier. In the viable hybrid model checker we implemented, symbolic methods are used to extract property-dependent functional information from the circuit-under-verification, and those informations are used to determine the *fitness function* and guide the simulation.

Our main contribution in this work lies in combining the advantages of both formal techniques and simulation-based methods in performing verification. A property-dependent logic partitioning approach using a fast preimage computation is first performed. The motivation for such a partitioning is to weed out "noise" embedded in the huge search space. In particular, correlations among state variables in the same partition will be favored over correlations among state variables in different partitions. An intelligent partitioning can uncover the correlations which are critical for verifying the target property, by putting closely related state variables into the same partition. Different from previously proposed techniques, where the property-independent partitioning is based only on the structural information of the circuit-under-verification, the new approximate preimage directed logic partitioning proposed in this work is property-dependent. Such a partition focuses

on the functional rather than the structural information. During stimuli generation, those dominant correlations will be emphasized (by assigning different weights to different partitions), such that the search for the target is better guided and the objective can be reached in a shorter time. The framework of our verifier has seamlessly integrated formal methods with simulation: approximate preimage in BDD format compactly represents the supporting logic of the target property, which determines the partitioned state space for the simulator to explore; on the other hand, a simulation-based state traversal overcomes the limit of formal methods on large designs due to the state explosion problem.

The remainder of the chapter is organized as follows. Section 6.2 presents preliminaries on problem formulation and partitioned state traversal. Section 6.3 gives the overall framework of our new method. Details on the proposed techniques are explained in Section 6.4. Experimental results are presented in Section 6.5, followed by the summary in Section 6.6.

6.2 Background and Motivations

A safety property expresses that something bad will never happen. Although simulation-based methods cannot prove correctness in verification, it fits quite well the objective of "debugging". In practice, for a large industry design, the verification objective is usually concerned more with finding bugs than providing formal proofs of correctness, especially in the early stages of the design. In other words, designers are often more interested in finding counter examples to the properties than proving it is 100% correct.

A safety property such as an invariant can be expressed as a Boolean expression on a subset of internal signals. Invariants hold in all reachable states of the design. We assume the properties referred in this work have been mapped to the state variables, where S_{target} represents the set of states that can violate the property. Subsequently, safety property checking becomes justification of S_{target}, where a search is performed to find an input sequence $I_0, I_1, ..., I_n$ that can reach S_{target}

from a given valid starting state. This process can be viewed as an ATPG procedure attempting to justify a target state. If the justification process succeeds, the property is falsified. Otherwise, when no input sequence can reach the target, the design is safe with respect to the property.

Before we delve into the new techniques, some definitions used in this chapter are listed below:

search space: state space consisting of all the state variables in the circuit-under-verification;

target space: a property-dependent state space consisting only of the state variables related to a property;

lighthouse: an indicator which is necessary to be true before or during the target is reached.

Lighthouses (LH) are monitored to ensure that the progression of the search is directed in the right direction, and they can play a role more than just a guide-post as pointed out in [70]. In our work, upon every move, the lighthouse “illuminates” to pick interesting vectors that is more likely to hit the target eventually.

6.3 Our Framework

The basic flow of the proposed hybrid model checker is shown in Figure 6.1. First, the target property is analyzed to partition the state variables in the search space. For each variable related to the target property, the approximate preimage for that variable is computed quickly; the shaded blocks in the figure denote formal techniques are used. Dominant logic correlations among FFs that can support a target bit (a target variable with its target logic value) are extracted individually to update the logic correlation matrix. All the state variables in the search space are partitioned based on this logic correlation matrix. Secondly, random vectors are generated to derive a partitioning on the target space. Information embedded in the set of reached states by the random vectors is used to partition the state variables related to the target property. A Genetic Algorithm is used

to generate test vectors on the partitioned state machine, whose fitness function contains both the fitness on the search space as well as on the target space. The population for the first generation of the GA is randomly generated, which contains n sequences of length λ. The GA iterates over m generations. The offsprings whose fitness values are above a threshold from generation $i-1$ are passed to generation i. After m generations, the vector sequence with the best fitness is selected and appended to the verification test set V. This process continues until the number of vectors in V reaches a predefined size. The *Genetic Spot Optimization* (GSO) block in the figure can be regarded as some coverage directed feedback mechanism, which dynamically adjust the fitness function of GA. The search also keeps track of the number of matched bits on very vector, and the probability profile of the vector sequence which leads the search to match more target bits is reused by GA.

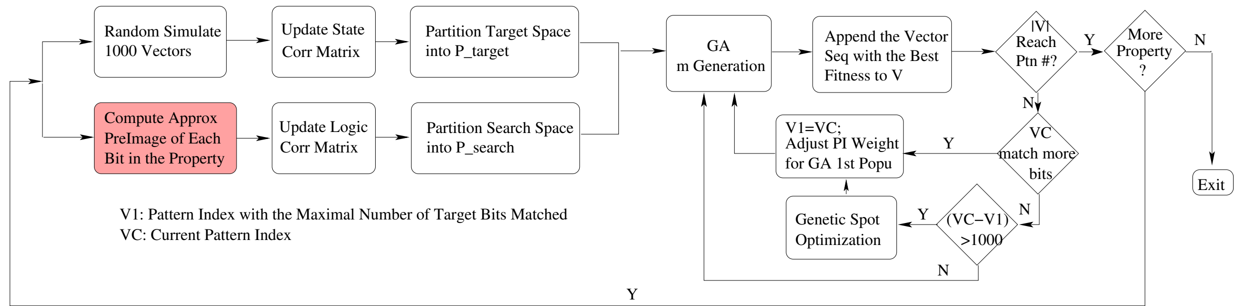


Figure 6.1. Framework of the Proposed Hybrid Model Checker

6.3.1 Fitness Function for GA

Shown in Equation (6.3), the overall fitness for a vector sequence consists of two parts: fitness on the search space and that on the target space. Fitness computation on the search space is shown in Equation (6.1) and (6.2), where $T_i(j)$ denotes the fitness for the ith partition on the jth vector, and W_i denotes the weight assigned to the ith partition. $T_i(j)$ is 10 if the reached state s brings a new partial state to partition i; otherwise, $T_i(j)$ is $1.0/F(i,s)(F(i,s) > 0)$, where $F(i,s)$ is the

number of the times partial state s has been reached on partition i. The sum of $T_i(j)$ along the vector sequence on all partitions forms the fitness of the vector sequence, where λ is the length of the vector sequence. Under this fitness function, new states will be favored by the GA.

Computation of fitness value on the target space differs a little bit from that on the search space. If state s illuminates the lighthouse H_i (H_i=1) on partition i, the fitness value for the individual would be assigned 10, but $F(i,s)$ is not incremented in the partial state history table. (The injection of lighthouse H_i on each partition of the target space will be explained in details in later sections.) This forces the subsequent search to view s as a new partial state and the GA will still favor state s. In other words, the fitness value for partial states s that can illuminate a lighthouse will not decay simply because it has been visited.

$$T_i(j) = \begin{cases} 10 & \text{if } s \text{ is new} \\ 1.0/F(i,s) & \text{otherwise} \end{cases} \tag{6.1}$$

$$F(i,s) = \begin{cases} 0 & T_i \text{ on Target space \& } H_i\text{=1} \\ F(i,s)+1 & \text{otherwise} \end{cases} \tag{6.2}$$

$$F_{total} = \overbrace{\sum_{\forall T_i \in S_{search}} W_i * \sum_{j=1}^{l} T_i(j)}^{F_{search}} + \overbrace{\sum_{\forall T_i \in S_{target}} \sum_{j=1}^{l} T_i(j)}^{F_{target}} \tag{6.3}$$

6.4 Our Approach to Verification

Our work focuses on developing some techniques to give better guidance to the logic-simulation-based pattern generator during the state traversal. The novelty of our work lies in two folds: (1) seamlessly combine formal methods with the simulation engine; and (2) propose a strong coverage metric directed feedback mechanism.

6.4.1 Approximate Preimage Directed Logic Partitioning

Structure-based partitioning has been proposed in literature for partitioned state traversal. In [71], controllability measurements are used for partitioning the FFs; FFs whose controllabilities are within the same range are put into the same partition. In [60], correlations are computed by counting the number of common paths for each pair of FFs; FFs with strong correlations are put into the same partition. Different from previous work, where partitioning is performed independent of properties to be verified, this chapter proposes a property-dependent partition method, based on the logic that can best support(satisfy) the target property.

6.4.1.1 Approximate Preimage Computation

The BDD is a compact form to represent the logic precisely. It may experience the state explosion problem when the number of variables is large. However, we do not need the exact preimage, since we are interested more in the correlation among state variables. Thus, approximation may suffice. Many approximation methods have been proposed in the past, such as [72, 73, 74], such that the size of the BDD may be reduced with the least amount of logic information loss. Some well-known BDD approximation methods include heavy branch subsetting and short paths subsetting.

To verify a safety property (to justify S_{target}), formal methods can directly perform backward reachability analysis by computing the preimage $PreImage(S_{target})$ iteratively until a fixed point is reached. If $PreImage(S_{target})$ has no intersection with the known initial state set, S_{target} is unreachable from any of the initial states. On the other hand, if $PreImage(S_{target})$ intersects with the initial state set, the property is falsified. The preimage represents the solution space for supporting S_{target}. One-cycle preimage is simply computed via Equation (6.4), where u, x, y, T and $S(y)$ represent the set of input variables, current state variables, next state variables, the transition relation and the target state, respectively.

$$Pre(T(u,x,y),S(y)) = \exists u, y \prod_{1 \leq i \leq k} T_i(u,x,y) \cdot S(y) \tag{6.4}$$

State explosion frequently occurs in intermediate BDDs during conjunctions in preimage computation. To overcome this, we split the transition relation T of the property and compute the preimage $Pre(T_i(u,x,y),S(y_i))$ for each single bit in the property individually, shown in Equation (6.5).

$$Pre(T_i(u,x,y),S(y_i)) = \exists u,y \;\; T_i(u,x,y) \cdot S(y_i) \tag{6.5}$$

Equation (6.6) inspires us that instead of extracting the supporting logic information embedded in the solution space from $Pre(T(u,x,y),S(y))$, another way is to individually extract those information from $Pre(T_i(u,x,y),S(y_i))$ and sum them up somehow.

$$Pre(T(u,x,y),S(y)) \subseteq \bigcap_{1\leq i\leq k} Pre(T_i(u,x,y) \cdot S(y_i)) \tag{6.6}$$

If state explosion occurs within the computation of Equation (6.5), the following short path based approximation algorithm is applied to iteratively pick the largest cube from the precise BDD until the predefined size limit is met. The sum of those cubes is an approximation of the original BDD. Procedure *Compute_Approx_BDD* in Figure 6.2 assumes that the circuit is levelized; “BDD” and “AppBDD” are the abbreviations for the precise BDD and its approximation.

6.4.1.2 Logic Correlation Extraction

After $Pre(T_i(u,x,y),S(y_i))$, the preimage/approximate preimage for each target bit in the property, is computed, analysis is performed on $Pre(T_i(u,x,y),S(y_i))$ to extract the dominant logic correlations for supporting this bit, and the correlation is represented in a square matrix. The extraction is performed by iteratively picking the largest cube C_j (where j denotes the index of the iteration) from $Pre(T_i(u,x,y),S(y_i))$ and universally quantifying the variables in C_j. Since a cube C_j can solely justify the target bit y_i, variables in C_j are highly correlated. We thus assign a positive value, 10, to each pair of FFs in C_j, representing the positive correlations between them. On the other hand, variables in different extracted cubes from different iterations (for example, variables in C_j and C_k) have little relationship in justifying y_i together because universal quantification guarantees to decouple variables in different cubes, and a negative value, -5, is assigned to each pair of FFs across different largest cubes, representing the negative correlations between them.

```
Procedure Compute_Approx_BDD(GateID)
GateID.BDD = ConstantOne;
GateID.AppBDD = ConstantZero;
∇ = GateID.Logic;
for (i=0; i< GateID.nFanIn; i++)
    ParentID = GateID.iFanIn[i];
    #check whether the BDD for the parent exists
    if (ParentID.BDD == NULL)
        ParentID.BDD = Compute_Approx_BDD(ParentID);
    GateID.BDD = GateID.BDD ∇ ParentID.BDD;
    if (Size(GateID.BDD) ≥ Size_Limit)
        while (Size(GateID.AppBDD) ≤ Size_Limit)
            LargestCube = PickLargestCube(GateID.BDD);
            GateID.AppBDD += LargestCube;
            GateID.BDD -= LargestCube;
        GateID.BDD = GateID.AppBDD;
return(GateID.BDD);
```

Figure 6.2. Algorithm for BDD Approximation

```
Proc Extract_Logic_Correlation(i)
j=0;
while (P_{i,j} != ∅)
    C_j = PickLargestCube(P_{i,j});
    for (k=0; k <|C_j|; k++)
        #perform universal quantification on variable v_k
        P_{i,j} = ∀v_k P_{i,j} (v_k ∈ C_j);
    P_{i,j+1} = P_{i,j};
    j = j+1;
UpdateLogicCorrMatrix();
```

Figure 6.3. Algorithm for Extracting Logic Correlations

The extraction Procedure *Extract_Logic_Correlation* is explained in Figure 6.3. In the procedure, i represents the index of the transition relation; j represents the index of the quantification iteration. We give an example to illustrate the iterative procedure. Assume there are 10 state variables (a, b, c, d, e, f, g, h, i, j) in the circuit. The target state to be verified is "XXXX10X1XX"; in other words, e=1(FF #5), f=0(FF #6), and h=1(FF #8). The approximate preimage for each bit is computed as $Pre(T_5(u,x,y)S(y_5{=}1))$, $Pre(T_6(u,x,y)S(y_6{=}0))$ and $Pre(T_8(u,x,y)S(y_8{=}1))$. Take FF #5 as an example, and assume $Pre(T_5(u,x,y)S(y_5{=}1))$ is supported by variables $\{a, b, c, d, e, f, g\}$. Starting from iteration j=0, $P_{5,0}{=}Pre(T_5(u,x,y)S(y_5{=}1))$, the largest cube C_0 is extracted, in which variables a and b are included. $P_{5,1}$ is obtained by universally quantifying variable a and b from $P_{5,0}$. In iteration j=1, the largest cube C_1 is extracted, which consists of variable e, f and g. $P_{5,2}$ is obtained by universally quantifying variable e, f, and g from $P_{5,1}$. Since $P_{5,2}$ is $\emptyset$, the extraction procedure for bit #5 is completed. Other bits in the property can be processed in the same way.

6.4.1.3 Logic Correlation Based Partitioning

Once the largest cubes have been computed as described, a matrix L is formed to record the logic correlations among each pair of state variables in the search space. Positive entries in the matrix represent logic correlations exist between two FFs in supporting the property; larger positive values denote stronger logic correlations. Negative entries in the matrix represent weak logic correlations exist between two FFs in supporting the property; larger negative values denote weaker logic correlations. A very large negative entry $L[i][j]$ indicates that state variables whose indexes are i and j can be safely decoupled in supporting the target property.

Continuing with the running example in Section 6.4.1.2. Matrix L initially contains all 0 entries. Figure 6.4 shows the entries in L at the end of each iteration after Procedure *Extract_Logic_Correlation* is called on state variable e(FF #5). Intelligent partitioning aims at emphasizing the dominant logic correlations among the state variables for supporting the property, weeding out the noisy correlations, and guiding the search efficiently in search space. Variables with strong logic correlations are placed into the same partition such that the correlations become intra-partition correlations, which will be favored by GA during the search. Variables with weak logic correlations are put into different partitions such that the correlations become inter-partition correlations, which will not be favored by GA during the search. If the final value of L is what is shown in Figure 6.4 and the size of each partition is 3, a partition set following the above partition criteria could be $\{a,b,c\}$, $\{e,f,g\}$, $\{d,h,i\}$, and $\{j\}$.

6.4.2 LH & GSO Based Feedback Mechanism

Feedback control has shown to be very useful in stimuli generation [61]. The basic idea is to extract useful information from the previous iterations to direct the search/prune the search space in the current iteration. Useful information extracted from previous iterations can accelerate the search procedure. In this work, we propose the concept of lighthouses and Genetic Spot Optimization

	a	b	c	d	e	f	g	h	i	j
a	0	10	0	0	0	0	0	0	0	0
b	10	0	0	0	0	0	0	0	0	0
c	0	0	0	0	0	0	0	0	0	0
d	0	0	0	0	0	0	0	0	0	0
e	0	0	0	0	0	0	0	0	0	0
f	0	0	0	0	0	0	0	0	0	0
g	0	0	0	0	0	0	0	0	0	0
h	0	0	0	0	0	0	0	0	0	0
i	0	0	0	0	0	0	0	0	0	0
j	0	0	0	0	0	0	0	0	0	0

iteration 0: C_0 = {a, b}
L[1][2] = L[2][1] = 10

	a	b	c	d	e	f	g	h	i	j
a	0	10	0	0	–5	–5	–5	0	0	0
b	10	0	0	0	–5	–5	–5	0	0	0
c	0	0	0	0	0	0	0	0	0	0
d	0	0	0	0	0	0	0	0	0	0
e	–5	–5	0	0	0	10	10	0	0	0
f	–5	–5	0	0	10	0	10	0	0	0
g	–5	–5	0	0	10	10	0	0	0	0
h	0	0	0	0	0	0	0	0	0	0
i	0	0	0	0	0	0	0	0	0	0
j	0	0	0	0	0	0	0	0	0	0

iteration 1: C_1 = {e, f, g}
L[5][6] = L[5][7] = L[6][7] = 10
L[6][5] = L[7][5] = L[7][6] = 10
count correlations between C_0 and C_1
L[1][5] = L[1][6] = L[1][7] = –5
L[5][1] = L[6][1] = L[7][1] = –5
L[2][5] = L[2][6] = L[2][7] = –5
L[5][2] = L[6][2] = L[7][2] = –5

Figure 6.4. An Example on the Construction of Logic Matrix

based feedback mechanism to efficiently accelerate the search.

6.4.2.1 Separating Search Space and Target Space

As shown earlier in Figure 6.1, different partition strategies are applied on the search space and the target space. This is based on the motivation that different state spaces play different roles in verifying the property. In the search space, partitioning is aimed at pruning the search space. To filter "noise" embedded in the search space and uncover useful correlations for verifying the property, formal partitioning is carried out on the search space, which is performed by considering

the logical correlations among FFs for supporting the target bits in the property.

The target space is mainly used to supply feedback information, where lighthouses are injected and Genetic Spot Optimization is carried out. A different partitioning criteria is applied on the target space: state variables whose target value pattern is some hard-to-justify portion of the property are put into the same partition. This forces the search to take the hard-to-justify portions of the property into consideration. Thus the following scenario can be suppressed — it is easy to satisfy portions of the property, while the search has no idea on how to glue different portions of the property together to achieve the verification goal.

6.4.2.2 Lighthouse Injection on Target Space

The concept of lighthouse is first proposed in [75]. Based on the observation that before a target can be verified, there may exist several levels of conditionals along different time frames which need to be satisfied, those conditional hints associated with a property are defined as "lighthouses". Lighthouses are actually Boolean variables, and SIVA proposed in [75] tries to justify the lighthouses together with the target along the forward simulation. If a lighthouse is justified, it ensures the progression of the search is in the right direction.

We borrow the concept of "lighthouse" from [75], but use it in a different way. On the partitioned target space, a lighthouse is injected on each state partition to check whether the partial state on a partition agrees with that portion of the property; if it agrees, the value of the lighthouse becomes logic 1 and it is said that the lighthouse on that partition is illuminated. For example, assume the target property to be checked is "X111010100". FF #2(b), #3(c) and #4(d) are in partition 1; FF #5(e), #6(f), and #7(g) are in partition 2; and FF #8(h), #9(i) and #10(j) are in partition 3. Lighthouses H_1, H_2 and H_3 are injected on the corresponding partitions: H_1=b&c&d, H_2=$\bar{e}$&f&$\bar{g}$, and H_3=h&$\bar{i}$&$\bar{j}$. When the logic value of H_1, H_2 or H_3 is true, the lighthouse on that partition is illuminated. Clearly, when all the lighthouses are illuminated, the target property is satisfied. Lighthouses are most useful to the search if they can catch hard-to-satisfy portions of the property,

thus it explains the partition strategy on the target space in Section 6.4.2.1. Lighthouses work closely with the Genetic Spot Optimization technique discussed in the next subsection to serve as a coverage metric directed feedback mechanism during the search.

6.4.2.3 Genetic Spot Optimization

In our method, the search keeps track of the number of matched target bits. V_c represents the current vector, with T_c as the number of bits matched. The index of the vector with the maximal number of matched target bits T_{max} is recorded in V_{max}. If $T_c > T_{max}$, the probability profile of the vector sequence from V_{max} to V_c is used by the GA as the profile for generating random vector sequences for GA's first population. This helps the GA to prune the search space before its first generation starts and direct the search to approach the target closer. If the maximal number of matched target bits keeps increasing, the search is proceeding in the right direction; if that number gets stuck, the search is trapped into some local optimum, which means it is trapped at a state closest to the target state and has a fitness higher than all its neighbors.

To solve the local optimum problem, we employ the concept of genetic spot optimization [66], which dynamically adjusts the fitness function (optimization goal) during the search. Now the goal of the search is not to satisfy as many bits in the target property as possible, but to satisfy hard-to-satisfy portions of the property first. The optimization spot consists of those partitions selected to be in the objective of the optimization. In other words, the optimization spot consists of those partitions in the target space that would affect the fitness value of a move. Partitions that belong to the target space, but not in the optimization spot, will not contribute to the fitness value, and the GA will not put optimization efforts on those partitions.

For each partition i in the target space, a counter C_i is set to record the number of times that the lighthouse H_i has been illuminated during the search. A partition with a big C_i is some easy-to-satisfy portion of the property; a partition with a small C_i (an extreme case is when C_i is 0, which indicates the lighthouse on the partition has never been met yet) is some hard-to-satisfy

```
Procedure Genetic_Spot_Optimization(cut_off)
#mark every partition not in the optimization spot
Reset_Partition_InSpot_Sign(D);
nSpot_Partition = 0;
max = Pick_Max_Value(C);
for (i=0; i<CIRT_nTarget_Partition; i++)
   if (C[i] <cut_off * max)
      D[i] = 1;
      nSpot_Partition++;
return(nSpot_Partition);
```

Figure 6.5. Algorithm for Genetic Spot Optimization

portion of the property. A portion of the property can appear to be easy-to-satisfy under one search direction, but hard-to-satisfy under the other search direction. When the search is stuck somewhere, Procedure *Genetic_Spot_Optimization*, shown in Figure 6.5, is invoked. The procedure reevaluates how difficult it is to satisfy different portions of the property under the current search direction (manifested by the probability profile of the vector sequence for GA's first population), by comparing the value of C_i on all partitions in the target space. Only those partitions whose C_i are below certain cut_off values are selected to be in the genetic optimization spot; other partitions are ignored during the optimization. Also, only the partitions within the genetic optimization spot account for the number of matched target bits T_c and T_{max}.

From the above procedure, it is seen that the genetic optimization spot can expand, shift or shrink during the search, together with the guidance from the lighthouses. After a new genetic optimization spot is obtained, the optimization objective and the search direction are readjusted accordingly.

6.5 Experimental Results

The proposed hybrid verifier is implemented in C. It was tested on both ISCAS89 and ITC99 benchmark circuits. Some of these benchmark circuits have large numbers of FFs, which can potentially make the formal verification methods that involve BDDs difficult to handle. The safety properties to be verified are random conjunctures of some FFs in the circuits.

First, we compare the verification results when the feedback mechanism is completely turned off/partially turned off/turned on. In Table 6.1, the first 4 columns describe the characteristics of the circuit, where *CKT*, *GATE #*, *FF #* and *Prop #* represent the name of the circuit, the number of gates in the circuit, the number of FFs, and the number of properties to be verified, respectively. The columns under *Option I* have both GSO and lighthouse injection disabled; the columns under *Option II* report the results with GSO disabled but LH enabled; the columns under *Option III* have LH enabled but GSO disabled; finally, the columns under *Option IV* report the results where both GSO and LH are enabled. For each circuit, under each option, the number of verified properties and the elapsed time in seconds are reported under the columns *verified* and *time*, respectively.

From Table 6.1, it is observed that the feedback mechanism performs the best when both GSO and LH are enabled. For the smaller circuits, all four options were able to verify all the properties. However, for the larger circuits, Option IV was able to perform much better. For all the 6 circuits tested, Option IV achieved the highest verification coverage in the least amount of time. Take b13 as an example. Among the 43 properties to be verified, under the proposed feedback mechanism (option IV), 42 of them were verified in 768.23 seconds. When either LH or GSO was turned off, 41/41 of them were verified in 1221.23/1050.20 seconds, which are 1/1 less in 453.00/281.97 more seconds. When both LH and GSO were turned off, 40 of them were verified in 1470.16 seconds. This demonstrates that the LH and GSO based feedback mechanism is very efficient, which brings significant performance improvement.

Table 6.1. Verification Results with/without GSO and/or LH Injection

CKT	GATE	FF	Prop	Option I Without GSO or LH		Option II With LH, Without GSO		Option III With GSO, Without LH		Option IV With GSO and LH	
	#	#	#	verified	time	verified	time	verified	time	verified	time
b08	189	21	47	**47**	813.06	44	1214.37	**47**	820.44	**47**	**686.66**
b12	1120	121	2	**2**	3.57	**2**	6.02	**2**	2.40	**2**	**1.16**
b13	362	53	43	40	1470.16	41	1221.23	41	1050.20	**42**	**768.23**
b19	105345	6618	60	**60**	28541.23	**60**	28040.86	**60**	37958.23	**60**	**27019.65**
s1423	753	74	43	28	9420.15	17	10320.61	29	9132.59	**39**	**8200.52**
s35932	39094	1728	8	0	48319.33	2	38257.02	2	41156.32	**3**	**35567.90**

We also compared our results with three other methods: THRIVE [69], VIS [76], and zChaff [77], reported in Table 6.2. THRIVE is a successful fault-simulation based model checker for safety properties. VIS is an efficient symbolic model checking tool for verifying properties. zChaff is a state-of-the-art SAT solver, which achieved significant performance gains over the conventional SAT solvers. For zChaff, we explicitly and incrementally unrolled the circuit 1 time frame at a time until either the property fails (SAT) or some resources (time/memory) run out. Experiments for all 4 model checkers were conducted over a 2.8 G Pentium 4 with 1G RAM, running on Red Hat Linux.

Table 6.2. Verification Results Compared with Other Model Checkers

CKT	GATE	FF	Prop	THRIVE [69]		VIS [76]		zChaff [77]		Our Approach	
	#	#	#	verified	times	verified	times	verified	times	verified	times
b08	189	21	47	**47**	800.13	**47**	**4.7**	**47**	141.03	**47**	686.66
b12	1120	121	2	**2**	3.22	**2**	2.2	**2**	34.51	**2**	**1.16**
b13	362	53	43	42	1250.20	**43**	**16.2**	40	3502.25	42	768.23
b19	105345	6618	60	**60**	43267.39	0	MEMOUT	30	89583.65	**60**	**27019.65**
s1423	753	74	43	32	9232.61	4	MEMOUT	38	19033.38	**39**	**8200.52**
s35932	39094	1728	8	2	42518.82	2	MEMOUT	1	42971.58	**3**	**35567.90**

As shown in the table, for smaller circuits, such as b08, VIS performed the best, followed by zChaff, both of which were performing a complete search. For larger circuits, such as b19, s1423, s35932, VIS could not build the BDD for the state transition function due to memory limitation; zChaff also aborted on some properties which are relatively far from the initial state, when more

time frames were involved in unrolling the circuits. On the other hand, the performance of our hybrid verifier was very good on those circuits. Take s1423 as an example, which contains 753 gates and 74 FFs. For the 43 target properties, THRIVE took 9232.61 seconds to verify 32 of them; VIS failed to build the BDD for the entire reachable state space, and before the memory ran out, 4 of the 43 properties were verified; zChaff aborted on 5 of the properties, and it took 19033.38 seconds for the 38 verified properties; finally, our hybrid model checker can verify 39 properties in 8200.52 seconds, which is 6/35/1 more than THRIVE, VIS and zChaff, respectively. The same trend was seen on b19 with 105345 gates and 6618 FFs and s35932 with 39094 gates and 1728 FFs.

6.6 Summary

A novel hybrid verifier combining formal partitioning with logic-simulation to verify safety properties of large sequential hardware designs is proposed in this chapter. Spatial correlations among the state variables of the CUT are further investigated via a property-dependent formal-method-based partitioning. The salient features of the proposed algorithm include: (1) formal methods are integrated into simulation by forming intelligent partitions on the state variables based on the quickly extracted logic correlations from a fast approximate preimage computation, and (2) a novel feedback mechanism based on lighthouses and Genetic Spot Optimization is employed within a Genetic Algorithms framework to adjust the fitness evaluation dynamically during the search. Experimental results on large benchmark circuits have shown the promising use of this hybrid verifier.

Chapter 7

Energy-Efficient Logic BIST Based on State Correlation Analysis

Like Chapter 5 and Chapter 6, this chapter continues to investigate spatial correlations among FFs. However, the motivation of utilizing the spatial correlations becomes minimizing power consumptions other than detecting more faults/design errors. Moreover, no partitioning is performed in this chapter; scan chain reordering is applied instead to emphasize important spatial correlations.

In this chapter, we present a new low power BIST for sequential circuits. State correlation analysis is first performed on the flip-flop values in the relaxed, compacted sequence for the undetected faults to extract spatial correlations among the flip-flops. The extracted spatial correlation matrix not only provides additional metrics through which the scan order may be altered, but also allows us to omit some flip-flops in the scan chain and supplies guidance for adjusting the speed of scanning for the rest flip-flops inside the scan chain. By leaving flip-flops that need less control out of the scan chain, we can reduce transitions on those flip-flops, thereby reducing the overall power and energy. The omission of flip-flops is done in a way such that the fault coverage is unaffected. According to their properties indicated by the spatial correlation matrix, the remaining flip-flops are

put into two scan chains, one at the normal clock rate, the other at a slower clock rate. Switching activities for both shifting inside the scan chain and propagating inside the combinational part can be further reduced by this dual-speed design without sacrificing fault coverage. Furthermore, reordering of the flip-flops in the scan chain allows the generated patterns to be more compatible with the state sequence necessary for exciting the random-pattern-resistant faults. Our experiments show that the same or higher fault coverage can be achieved with less energy (and average power) - average power of 48.5% is reduced, with the maximum reduction of 73%.

7.1 Introduction

Power-dissipation issues are playing increasingly important roles in testing VLSI (very large scale integration) circuits. There are several factors leading to this situation. First, the growing size of VLSI circuits, high transistor density and popularity of SOC design make testing of VLSI circuits harder. Consequently, testing time and energy consumed during testing are increasing. Second, the rapidly increasing operating speed of the circuit-under-test (CUT) worsens the power consumption problem during testing, because higher frequency implies both heightened average and peak powers. Third, portable and wearable devices are designed for low power purposes, thus low power testing would be desirable. Finally, excessive power dissipation during testing can cause problems, many of which can permanently damage the CUT or make the CUT less reliable, e.g., elevated electromigration rate and increased power/ground noise.

Scan-based BIST architectures are popular nowadays. However, power consumption can be a greater threat since scanning in globally-unreachable states frequently results in additional switching activity. In this chapter, a partial-scan BIST architecture containing an LT-RTPG is proposed that significantly reduces power consumption. Scan chain alteration in scan chain length, scan chain partition and scan order is accomplished by state correlation analysis, in which a *state spatial correlation matrix* is constructed for a subset of hard faults in the circuit. Reducing length of

the scan chain and employing dual-speed design for scan chain helps to further reduce power and test application time, while reordering of the scan chain helps to compensate loss of fault coverage.

Our aim is to further reduce power dissipation while retaining the original fault coverage. A big difference our technique has with the previous schemes mentioned above is that we scan only those flip-flops that will help achieve our goals. Our work begins with the undetected fault set, after a random input sequence of certain length is first applied to the CUT. Then, we check to see if any of the undetected faults was ever excited; if so, we record the excitation segments for each excited fault in the undetected fault set. Test compaction similar to that described [78] is performed on the excitation sequence, followed by test sequence relaxation [46]. Both compaction and relaxation are performed to filter noise, leaving the final sequence with only the key and necessary information for exciting the target fault. We use the information embedded within the flip-flop values corresponding to the relaxed and compacted excitation sequences to construct *state spatial correlation matrix*, with which the flip-flop selection, partition and reorder of the scan-chain is performed.

As discussed before, most low power BIST are only applicable for combinational circuits or full-scan sequential circuits. Moreover, switching activity for most circuits is proportional to the switching activity inside the scan chain, and longer scan chains generally will incur more switching as non-scanned flip-flops are less controllable. Therefore, we carefully design the scan chain such that the flip-flops that never contribute to fault coverage are left out of the scan chain. Some flip-flops are included in the scan chain only for the purpose of propagating the fault effects, while controlling their values does not help in exciting hard faults. These flip-flops are called *propagating flip-flops*, and they can be put into a scan chain driven by a slower clock.

We also alter the scan chain order to help detect the missed "hard" faults. The alteration is obtained by utilizing the spatial correlation information of the internal states. The set of correlations of every two neighboring cells in the scan chain can be determined by the TPG that is feeding the scan chain; the correlation is in the form of *toggle probability* as indicated in Table 7.1. A second set

of correlations for two flip-flops necessary for exciting "hard" faults is contained in *state spatial correlation matrix*. By altering the scan chain order, we minimize the difference between the two sets of correlations, so that the patterns generated are more compatible with the state sequence necessary for exciting/detecting the hard faults. In this way, we increase the possibility for detecting hard faults and every switching inside the scan chain contributes more to fault excitation or fault detection. The resulting BIST architecture can be more energy efficient, that is, either more faults detected with the same amount of energy or the same fault coverage achieved in less energy.

The criterion here is that altering the design of the scan chain (including scan chain length reduction, scan chain partition and scan chain reorder) must not affect the fault coverage of scan testing. In other words, we are not trying to lower the average power at the reduction of fault coverage. Our technique inherits the merits of LT-RTPG [32], that is, negligible hardware overhead, purely random test input, and it can achieve improvements in both power consumption and fault coverage.

The remainder of the chapter is organized as follows. Section 7.2 gives an overview of LT-RTPG. Section 7.3 explains the construction of the *state spatial correlation matrix* via state analysis. Section 7.4 reports the experimental results, and Section 7.5 summarizes the chapter.

7.2 Preliminaries

Previously, LT-RTPG [32] has shown that it can reduce the heat dissipated during BIST session by significant amounts while attaining high fault coverage. The structure of LT-RTPG is shown in Figure 7.1. It is comprised of an LFSR of size r, a k-input AND gate ($k < r$), and a Toggle 'T' flip-flop. Compared with those obtained by the LFSR patterns in terms of the number of transitions and fault coverage, the results obtained by the LT-RTPG patterns are 14-37% less in the number of transitions, while suffering at most 3% less in fault coverage. Since heat dissipation in a CMOS circuit is proportional to switching activity, LT-RTPG presents an energy efficient BIST architecture.

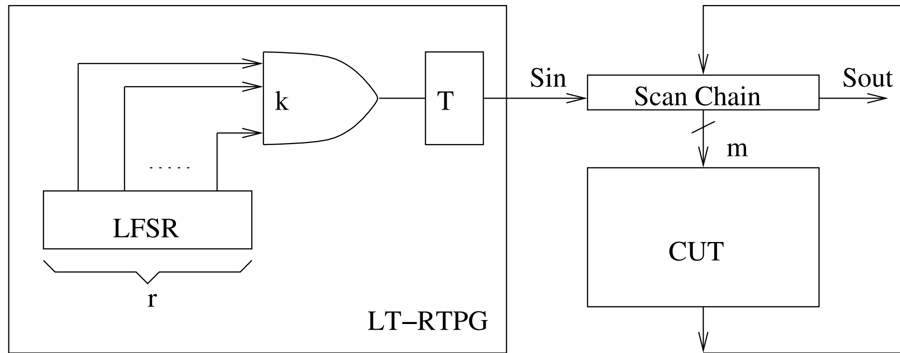


Figure 7.1. Architecture of LT-RTPG

7.2.1 Structure of LT-RTPG

LT-RTPG [32] is shown in Figure 7.1. A brief description is given below to illustrate the reason why the structure of LT-RTPG can reduce power dissipation while still retain high fault coverage. The scan chain is driven by the output of the TFF. The signal probability for each input of the AND gate is 0.5. Since a TFF holds its value until the input of the TFF is assigned 1, the same value is repeatedly scanned into the scan chain until all the inputs of the AND gate become 1. Hence, neighboring scan flip-flops are likely to be assigned identical values in most scanned test patterns. The probability that the output of the TFF toggles is determined by the signal probability of the output of the AND gate, which is simply $\frac{1}{2}^k$, where k is the number of inputs of the AND gate. The LT-RTPG reduces the number of transitions at the input of the scan chain by a factor of $2^{(k-1)}$ [32]. This reduction in the number of transitions is what enables the LT-RTPG to reduce power dissipation in the CUT during test application. On the other hand, assigning identical values to neighboring state inputs reduces the uniformity of distribution and may adversely affect the fault coverage. The properties of patterns generated by the LT-RTPG are analyzed in [32]. Let the probability that state input s_i is assigned the value v and state input s_j is assigned the opposite value, be called the *toggle probability* of s_j with respect to s_i. The *toggle probability* of LT-RTPG is shown in Table 7.1, where the term 'distance' gives the number of flip-flops between two specified

flip-flops along the scan chain. For example, if the order of the scan chain is 1->2->3->4->5->6, considering the distance of flip-flop 1 and 6, we notice that there are four flip-flops, 2, 3, 4, 5 between these two, so the distance should be 4+1 = 5. In order to make every possible pattern appear at a set of inputs with equal probability, the *toggle probability* of any two inputs should be 0.5. And this rule is satisfied when a scan chain is driven by a simple LFSR if an exhaustive sequence of patterns is applied. In contrast, the use of the LT-RTPG decreases toggle probability of neighboring state inputs and that can lead to loss of fault coverage. Although LT-RTPG is test-per-scan BIST, it allows at-speed testing. If the length of the scan chain is N, we need (N+1) cycles to apply an input pattern. In the N shift-in cycles, we analyze both POs and the scan out value for fault detection, while in the capture cycle, we only consider POs for fault detection.

Table 7.1. Toggle Probability

distance	*k*		
l	2	3	4
1	.250	.125	.063
2	.375	.219	.117
3	.438	.289	.165
4	.469	.342	.207

7.3 Computing State Spatial Correlation Matrix

As we pointed out in the former section, the LT-RTPG can achieve lower power dissipation, but may bring in loss of fault coverage. To describe how power efficient the BIST architecture is, we define *energy efficiency* as the ratio between fault coverage and total number of switching activities during the application of test patterns. To make a fair comparison, we compute *energy efficiency* of different BIST architectures for the same CUT by applying test patterns of the same length.

$$EE = \frac{fault\ coverage}{total\ switching\ activity\ during\ testing} \tag{7.1}$$

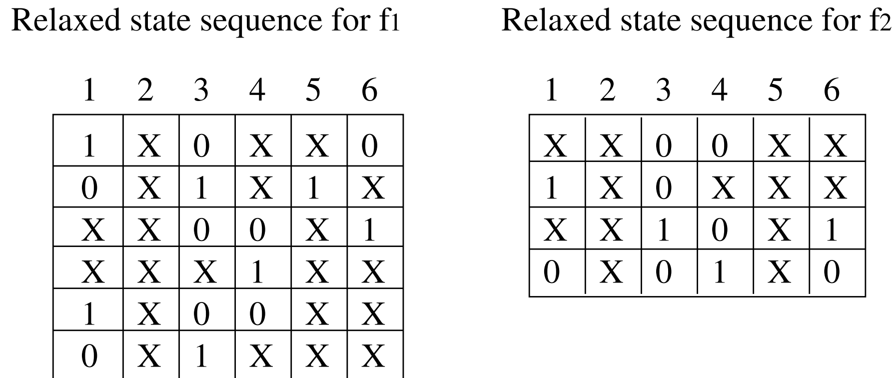
It is obvious that higher *energy efficiency* results in better architecture for low power purposes. We define *state spatial correlation matrix* to help improve *energy efficiency*. The elements inside the state spatial correlation matrix indicate either (1) how important an FF should be controlled in order to excite hard faults or (2) the necessary spatial correlation between two FFs in exciting hard faults. Using the measurements provided by the matrix, we want to reorder the scan chain to minimize the difference of the sets of spatial correlations for any pair of neighboring FFs inside the scan chain between the two views (the TPG's view and the CUT's view), so that every switch will contribute more to fault excitation or fault detection and the system has a higher *energy efficiency*.

7.3.1 Underlying Methodology

An example illustrating how and why our technique works is given below. We first fault simulate a set of random vectors on the CUT without scan. Assume that only two faults f_1 and f_2 remain undetected after the simulation. We collect all the excitation sequence for each of the undetected faults, then static compaction and state relaxation are performed for every excitation sequence of a single target fault. We then concatenate the relaxed state sequences for each target fault. Figure 7.2 shows the relaxed state sequence for our example, where each row is a state vector, and 1, 0, X inside a block denotes logic 1, logic 0 and don't care. The following matrix shows

$$M = \begin{bmatrix} 0.60 & 0.00 & 0.83 & 0.33 & 0.17 & 0.17 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.55 & 0.00 & 0.90 & 0.22 & 0.00 & 0.11 \\ 0.33 & 0.00 & 0.33 & 0.60 & 0.00 & 0.50 \\ 1.00 & 0.00 & 0.00 & 0.00 & 0.10 & 0.00 \\ 0.25 & 0.00 & 0.25 & 0.75 & 0.00 & 0.40 \end{bmatrix}$$

the *state spatial correlation matrix M* extracted from the relaxed sequences in Figure 7.2 by using the algorithm given later in next subsection.



Relaxed state sequence for f1

1	2	3	4	5	6
1	X	0	X	X	0
0	X	1	X	1	X
X	X	0	0	X	1
X	X	X	1	X	X
1	X	0	0	X	X
0	X	1	X	X	X

Relaxed state sequence for f2

1	2	3	4	5	6
X	X	0	0	X	X
1	X	0	X	X	X
X	X	1	0	X	1
0	X	0	1	X	0

Number above the block indicates the index of the flip flop

Figure 7.2. Relaxed Excitation Seqs for Undetected Faults f_1, f_2

Essentially, this matrix shows either (1) how important an FF should be placed in the scan chain to be controlled in order to excite hard faults by its diagonal elements or (2) the necessary spatial correlation between two FFs in exciting hard faults by its non-diagonal elements. According to the *state spatial correlation matrix* M, we decide flip-flop #2 and #5 should be eliminated from the scan chain, because M_{22} and M_{55} are relatively small (0.0 and 0.1), showing that these two flip-flops seldom need to be controlled for exciting f_1 and f_2. Based on this observation, using our correlation matrix M, we can alter the scan chain by removing these 2 FFs and reorder the FFs according to their spatial correlation computed in the scan chain, which is shown in Figure 7.3.

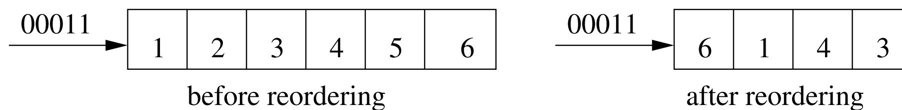


Figure 7.3. Comparison of Scan Chain before and after Altering

Two test patterns are said to be compatible if no conflict is found in logic values for every bit pair of these two patterns. For example, ”1X01” and ”X0X0” are compatible for bits #1, #2, and #3, but conflict arises for bit #4; so these two patterns are not compatible. Figure 7.4 compares the

compatibilities of shifted patterns with the relaxed sequences in Figure 7.2 before and after scan chain altering. The first row gives the initial state of the scan chain before shifting in pattern 11000.

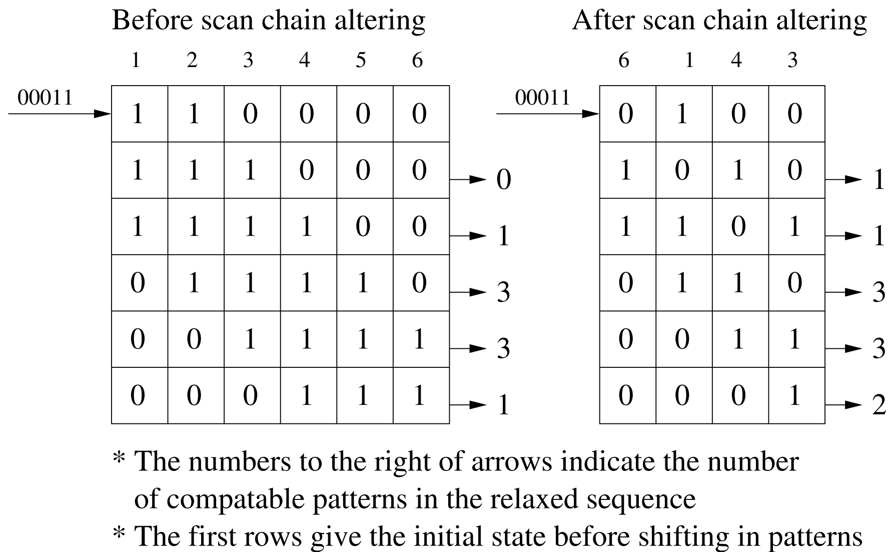


Figure 7.4. Compatibilities before and after Scan Chain Altering

The number pointed by an arrow on the right side of the block indicates how many patterns in the relaxed sequences are compatible with the current state of the scan chain. For compatibility, we will not consider the FFs that are not inside the scan chain. So after FFs selection, we need not consider the compatibility of FF #2, #5. The total number of compatibility before altering is 0+1+3+3+1=8, and that after altering is 1+1+3+3+2=10. The difference in compatibility in real case is actually higher, and we just want to use this example to show how the scan chain altering can help to raise the total number of compatibility and make the TPG generated patterns more compatible with the necessary exciting sequence for the hard faults of the CUT. The correlation sets in terms of *toggle probability* before and after scan chain reorder are shown in Figure 7.5. The difference of correlation sets between CUT's point of view and LT_RTPG's point of view for each scan chain structure is calculated by Equation (7.2), where l is the length of the scan chain and $M_{i,j}$

is the element on the ith row, j^{th} column of the *state spatial correlation matrix*.

$$\sigma = \frac{1}{0.25^2} \sum_{i=1}^{l-1} (M_{i,j} - 0.25)^2 \quad (7.2)$$

where i and j are neighboring FFs inside the scan chain

In the equation, 0.25 is the ideal *toggle probability* of two neighboring FFs from LT_RTPG's point of view if a 2-input AND gate is used to feed the TFF. For LT_RTPG using AND gate other than 2-input, check Table 7.1, and replace 0.25 by the corresponding value in the first row. We get the difference before scan chain altering is 4.01 and the difference after scan chain altering is 0.20. Obviously, the scan chain after altering fits the LT_RTPG better.

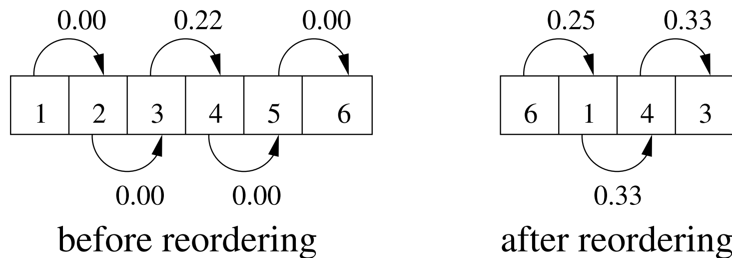


Figure 7.5. Toggle Probability before and after Scan Chain Altering

7.3.2 Algorithms

The entire procedure for our algorithm is outlined in Figure 7.6. Both compaction and state relaxation are used in the algorithm to filter noise and extract the necessary and key information as the characteristic of the CUT. The algorithm for **COMPUTE_CORRELATION** is shown below in Figure 7.7.

The *state spatial correlation matrix M* gives two measurements. First, the diagonal elements of the matrix indicate how often the FF should be set to a specific value for exciting hard faults. Larger value indicates the corresponding FF should be scanned to be able to control its value.

```
MAIN()
apply random test set T on circuit C;
U = undetected fault set;
R = ∅;
for each undetected but excited fault, f_i ∈ U
    fault simulate T on f_i to extract the excitation segments {E_{i,j}};
    R_i = compact and relax {E_{i,j}};
    append R_i to R;
M = COMPUTE_CORRELATION(R);
D = FILTER(M);
/* compute propagation matrix G */
for each undetected fault, f_i ∈ U that propagated to a FF j
    G[i][j] = 1;

/* rearrange scan chain based on D */
while (not done)
    L = REARRANGE(D, i);
    FC = fault coverage with this partial-scan BIST L;
if (max FC < full-scan LT-RTPG)
    D = CORRECTION(M) and repeat the while loop;
return L with max FC;
```

Figure 7.6. Algorithm for Configuring the Scan Chain

COMPUTE_CORRELATION($R[m][n]$)
Initialize matrix $M[n \times n]$;
Diagonal elements of M, $M[i][i] = \frac{number\ of\ fully\text{-}specified\ bits\ in\ column\ i\ of\ R}{sequence\ length(m)}$
Non-diagonal elements of M, $M[i][j](i \neq j) = \frac{number\ of\ conflicting\ bit\ pairs\ between\ column\ i\ and\ j\ of\ R}{M[i][i]}$
return(M);

Figure 7.7. Algorithm for Constructing Spatial Correlation Matrix

For those diagonal elements whose value is either zero or less than the selected cutoff value, their corresponding FFs can be kept out from the scan chain because their value are always don't care for hard fault excitation. The reduction in the length of the scan chain will contribute to the reduction of switching activity and test application time. The non-diagonal elements of the matrix indicate the *toggle probability* of two FFs in excitation sequence. And we hope the set of *toggle probability* of neighboring FFs in the scan chain decided by the LT-RTPG in Table 7.1 matches the profile of *toggle probability* provided by the matrix to increase the *energy efficiency*.

The algorithm for procedure **FILTER** is listed in Figure 7.8. It gets rid of those FFs which do not need to be scanned and returns to a new matrix which only consists of the information of FFs that are in the scan chain.

The algorithm for procedure **REARRANGE** is listed in Figure 7.9. The algorithm **REARRANGE** helps in reordering the FFs in the scan chain. The input parameter to the procedure is the *state spatial correlation matrix* after filtering, and the specified head FF of the scan chain. Starting from the head FF, the FF whose *toggle probability* with respect to the head FF is most close to 0.25 (when 2-input AND gate is used) is selected as the neighboring FF of the head FF. The newly selected FF is then acting as the base FF, and the same procedure for specifying neighboring FF for base FF goes on until every FF is fit into the scan chain. The procedure is easy to follow and

```
FILTER(M[n][n])
max = maximal diagonal element in M;
threshold = k% of max;
for (i=0; i<n; i++)
    if (M[i][i] > threshold)
        FF #i stays in the scan chain;
D = modified M by removing the eliminated columns and rows;
return(D);
```

Figure 7.8. Algorithm for Selecting Scan Flip-flops

is not globally optimized. To reduce area overhead by rearranging the scan chain, **REARRANGE** should take layout information into consideration; however, in this work, we did not realize that. If layout information is available, we can rearrange FFs in a multilevel fashion. FFs are first grouped according to their positions so that FFs inside a group are a cluster on layout. Procedure **REARRANGE** is applied on every group and then applied on the set composed of the boundary FFs of all groups. To further reduce area overhead, the size of each group should be restricted to be a small number, and rearrangement can be applied only inside each group but not among groups.

Considering the fact that some hard faults are not able to be propagated to a PO, we also need to re-insert some FFs which are originally filtered out by the procedure **FILTER**. To remedy this problem, in the algorithm **MAIN**, a propagation matrix G is constructed for this very purpose. In this matrix, an element $G[i][j] = 0$ means that fault j is not propagated to FF i; while $G[i][j] = 1$ indicates that fault j is propagated to FF i at least once. Thus, the sum of all the values in row i in G (SUM(G_i)) indicates the number of hard faults propagated to FF i. Higher value of SUM(G_i) indicates more faults can be detected if FF i is scanned. The algorithm **CORRECTION** re-inserts the needed FFs with better propagation abilities into the scan chain, L, and adjust matrix D to include information of those re-inserted FFs. The algorithm for **CORRECTION** is listed in Figure

REARRANGE($D[t][t]$, i)
$L = \emptyset$;
$L[0]$ = FF #i;
$t = 1$;
while (not done)
    select the next FF j such that $D[i][j]$ is optimal (closest to 0.25)
    $L[t++] = j$;
    i = j;
return(L);

Figure 7.9. Algorithm for Scan Chain Reordering

7.10.

7.4 Experimental Results

We conducted experiments using our approach on ISCAS89 [47] and ITC99 [48] sequential benchmark circuits. We allowed for a maximum of 10,000 random vectors for extracting the excitation sequence of undetected faults. (The number of initial random vectors should be determined by the size of the CUT.) *State spatial correlation matrix* is constructed after static compaction and state relaxation are performed on the excitation sequence of each undetected fault individually. For most of the circuits tested, CPU time to construct the *state spatial correlation matrix* is less than one minute. Some hard to test circuits such as s5378 may take up to an hour.

Table 7.2 shows the results of our approach, compared with the ordinary LFSR BIST, LT-RTPG BIST in terms of fault coverage, power dissipation and *energy efficiency*. In the table, the column FC, SW, EE are the abbreviation of fault coverage, switching activity in millions (weighted by number of gate fanouts) and *energy efficiency*, respectively. Energy efficiency has been defined in

```
CORRECTION(G[n][n])
for (all FFs not in the scan chain)
    SUM(G_i) = Σ_{∀j} G[i][j];
max = maximum SUM(G_i);
threshold = k% of max;
while (not done)
    if (SUM(G_i) > threshold)
        re-insert FF i to scan chain;
adjust filtered correlation matrix D with the newly inserted FFs;
```

Figure 7.10. Algorithm for Correction

Equation (7.1). The column FF% gives the ratio of the number of scanned FFs over the number of total FFs in the circuit for each technique. We take the switching activity and *energy efficiency* of ordinary LFSR as unit 1. Switching activity and *energy efficiency* of other architectures are all normalized with respect to those of ordinary LFSR. The row "average" gives the average for each column. For convenience, we shall call our technique LSB_RTPG and LSA_RTPG (Low-power-State-analysis-based-random-pattern-generator-*Before/After*-scan-chain-reorder) to represent before and after scan chain reorder, respectively. We did not include the model of MISR (multiple input signature register) in our simulation, and any conventional MISR can be added to work with our technique. Due to the fact that the number of scan-in cycles is reduced by partial scan, the energy consumed in the ordinary MISR during testing will be reduced at the same percentage.

From the table, we can see that LT-RTPG [32] reduces power between 13%-39% at a loss of fault coverage of at most 3%. The *energy efficiency* of LT-RTPG is higher than that of original LFSR BIST, ranging from 1.27-1.56 times of that of the original LFSR BIST. The fault coverage of our LSB_RTPG (before FF reordering) is the same or higher than that of LT-RTPG for all circuits except in circuits s444, s1196 and s5378; the energy reduction of 24.3% was achieved. When FFs

Table 7.2. Comparison of Fault Coverages, Heat Dissipation and Energy Efficiency

Ckt	Length	LFSR			TFF				Ours						
										before reorder			after reorder		
		FF%	FC	SW (M)	FF%	FC	SW	EE	FF%	FC	SW	EE	FC	SW	EE
s298	1024	14/14	**100.0**	1.29	14/14	96.43	.759	1.27	7/14	99.68	.321	**3.11**	99.35	.326	3.05
s344	2048	15/15	**100.0**	3.23	15/15	**100.0**	.692	1.45	7/15	**100.0**	.337	2.97	**100.0**	.332	**3.01**
s349	2048	15/15	**99.43**	3.26	15/15	**99.43**	.691	1.45	7/15	**99.43**	.337	2.97	**99.43**	.331	**3.02**
s386	2048	6/6	**98.18**	1.20	6/6	**98.18**	.782	**1.28**	6/6	**98.18**	.782	**1.28**	**98.18**	.782	**1.28**
s444	2048	21/21	**97.05**	4.49	21/21	**97.05**	.719	1.39	16/21	93.25	.492	1.95	94.09	.467	**2.07**
s526	4096	21/21	**99.28**	13.21	21/21	96.76	.655	1.49	14/21	97.12	.381	1.91	98.20	.399	**2.48**
s641	4096	19/19	98.50	7.76	19/19	98.72	.720	1.39	14/19	**99.36**	.568	**1.78**	**99.36**	.568	**1.78**
s713	4096	19/19	92.25	8.31	19/19	92.43	.726	1.38	14/19	**92.94**	.573	**1.76**	**92.94**	.573	**1.76**
s820	8192	5/5	**98.35**	9.76	5/5	97.06	.692	**1.43**	5/5	97.06	.692	**1.43**	97.06	.692	**1.43**
s832	8192	5/5	**96.55**	10.02	5/5	95.40	.689	**1.43**	5/5	95.40	.689	**1.43**	95.40	.689	**1.43**
s1196	4096	18/18	**96.54**	3.97	18/18	**96.54**	.872	1.15	8/18	96.05	.514	1.93	96.05	.469	**2.12**
s1423	4096	74/74	98.42	102.95	74/74	98.48	.643	1.56	67/74	98.81	.567	**1.77**	**99.01**	.631	1.59
s1488	4096	6/6	**99.80**	10.93	6/6	98.99	.700	1.42	6/6	98.99	.700	1.42	99.39	.636	**1.57**
s1494	4096	6/6	**99.00**	11.04	6/6	98.14	.701	1.41	6/6	98.14	.701	1.41	98.61	.674	**1.48**
s5378	65k	179/179	**99.02**	13911	179/179	98.57	.741	1.34	148/179	98.44	.595	**1.67**	98.96	.621	1.60
b01	256	5/5	**100.0**	0.042	5/5	97.78	.758	1.29	3/5	99.26	.538	1.84	**100.0**	.499	**2.00**
b06	256	9/9	**100.0**	0.102	9/9	**100.0**	.743	1.35	3/9	**100.0**	.297	3.37	**100.0**	.274	**3.65**
b07	8192	51/51	93.99	93.47	51/51	93.24	.634	1.56	43/51	95.09	.488	2.07	**98.39**	.461	**2.27**
b08	8192	21/21	96.52	17.47	21/21	93.66	.640	1.52	19/21	95.29	.565	1.75	**98.77**	.483	**2.12**
b12	16k	121/121	95.04	818.81	121/121	94.33	.615	1.61	109/121	95.58	.527	1.91	**97.29**	.471	**2.17**
average			97.90			97.06	.709	1.41	.755	97.40	.533	1.99	97.99	.515	2.09

FF%: % FFs scanned FC: Fault Coverage SW: Total Weighted Switching in Millions EE: Energy Efficiency

are reordered, LSA_RTPG achieves better results in both fault coverage and energy reduction than prior to FF reordering (LSB_RTPG). Of the total 20 circuits tested, the highest fault coverage via LSA_RTPG was achieved for 12 circuits. And for the other 8 circuits, 5 of them have higher fault coverage than that of LT-RTPG. The average *energy efficiency* for LSA_RTPG is 2.09, the highest among all techniques. Compared with LFSR, the average energy reduction is 46.7% and 48.5% in our technique for LSB_RTPG and LSA_RTPG, respectively. When compared with LT-RTPG, average energy reductions of 25% and 27% are achieved for LSB_RTPG and LSA_RTPG.

The procedure of filtering in our technique is able to eliminate up to 80% of FFs to be scanned, without loss of fault coverage. The average length of scan chain after filtering is 75.5% of that of full-scan. For example, compared with LT-RTPG, the length of the scan chain is decreased from 21 to 14 for s526 in our technique. This helps in further lowering switching activities since the non-scanned FFs now experience less activity than if they were kept in the scan chain. The switching activity after eliminating the FFs is lowered by an additional 42% percent, and the fault coverage increased from 96.76% to 97.12% for LSB_RTPG. In our technique, after scan reordering, almost every circuit gets improved in fault coverage. For example, the fault coverages of b07, b08 and b12 get improved from 95.09%, 95.29%, and 95.58% to 96.45%, 98.98% and 98.26% respectively, which are even higher than the ordinary LFSR BIST. Take b12 for example. Fault coverage of LT-RTPG is 94.33%, which is 0.71% lower than ordinary LFSR. After selecting 109 FFs to be scanned from the total number of 121, without reordering the FFs, we increase fault coverage in LSB_RTPG to 95.58%, which is 0.54% higher than that of the ordinary LFSR, and 1.25% higher than that of LT-RTPG. And we are able to further reduce energy by 14.31% with respect to LT-RTPG. After reordering the FFs in the scan chain, LSA_RTPG improves fault coverage to **98.26%**, which is 2.68% higher than before reordering the FFs, while the energy consumed is also lower than that before reordering. So after reordering, compared with ordinary LFSR, *energy efficiency* of b12 is more than 2.0 and fault coverage is improved by 3.22%.

7.5 Summary

We have presented a new and effective approach for improving both the fault coverage and power (energy) dissipated for logic BIST by utilizing the spatial correlations among FFs. State analysis is performed on the compacted relaxed excitation sequence of the hard faults. The extracted information is stored in the *state spatial correlation matrix*, which is used later for matching the correlation set from the TPG's point of view with the correlation set from the CUT's point of view to make the entire system more energy efficient. Experimental results show that *energy efficiency* is increased significantly without loss of fault coverage for most circuits (sometimes fault coverage is even increased). An average of 48.5% energy reduction is achieved, with the maximum reduction of 73%.

Chapter 8

Conclusion

In this dissertation, we addressed three related simulation-based pattern generation problems in Verification and Testing: sequential ATPG targeting SSFs, safety property verification, and low power testing for full-scan sequential circuits. We introduced spectral analysis, characteristic faults and characteristic states to help catch temporal correlations on PIs. We also proposed state traversal on partitioned state space and state correlation analysis to better catch spatial correlations among the state variables. All these correlation-based heuristics aim at generating high quality simulation vectors in a more efficient manner.

First, we presented an effective approach for logic BIST using characteristic faults and temporal correlations. We demonstrated that the temporal correlations embedded in the spectrum of the sequence that can detect the characteristic faults can also detect many other faults in the CUT. After we spectrally characterize the circuit, this information is sufficient in aiding BIST to obtain extremely high fault coverages. We further extended this temporal correlation based BIST architecture to test a cluster of sequential cores on the SOC in parallel, which speeds up the test application and reduces the test cost.

Secondly, we utilized spatial correlations among the state variables of the CUT via partitioning on those variables. Partitioning on the state variables can emphasize some spatial correlations, but filter the rest. Different partitioning strategies have been proposed to be able to catch the important spatial correlations that are closely related to fault/error detection. A structure-based partition method, together with a cooperative game based traversal has been proposed to exploit the orthogonality on two global partition sets to enhance the test generation. Experimental results show that test vectors generated by the proposed method can achieve an average of 10% higher coverage for design validation. Furthermore, a property-dependent formal partition method has been proposed to catch the key spatial correlations among FFs in supporting the given property logically. This results in a salient hybrid verifier which seamlessly combines formal methods with logic simulation. Experimental results have shown the promising use of this hybrid verifier in verifying the safety properties of large sequential hardware designs.

Finally, spatial correlations among FFs are extracted via state correlation analysis for improving both the fault coverage and power (energy) dissipated for logic BIST. The extracted spatial correlations are stored in the *state spatial correlation matrix*, which is used later for matching the correlation set from the TPG's point of view with the correlation set from the CUT's point of view to make the entire system more energy efficient. Experimental results show that an average of 48.5% energy reduction is achieved, with the maximum reduction of 73%.

This work suggests future directions of research. Taking into account the vast amount of possible correlations among circuit variables, from a practical point of view, efficient data mining techniques can significantly improve the quality of the extracted correlations. Thus those useful temporal/spatial correlations can better guide the simulation-based sequential ATPG engine.

Bibliography

[1] Synopsys, "TetraMAX ATPG," in *http://www.synopsys.com/products/test/tetramax_wp.html*.

[2] J. Roth, "Diagnosis of Automata Failures: a Calculus and a Method," *IBM Journal of Research and Development*, vol. 10, pp. 278–291, July 1966.

[3] V. Agrawal, C. Kime, and K. Saluja, "A Tutorial on Built-In Self-Test," *IEEE Transactions on Design & Test of Computers*, vol. 10, pp. 73–82, April 1993.

[4] P. Bardell, W. McAnney, and J. Savir, *Built-in Test for VLSI, Pseudorandom Techniquesn*. John Wiley & Sons, 1987.

[5] B. Nadeau-Dostie, D. Burek, and A. Hassan, "ScanBist: A Multifrequency Scan-Based BIST Method," *IEEE Transactions on Design & Test of Computers*, vol. 11, pp. 7–17, January 1994.

[6] X. Chen and M. Hsiao, "Characteristic Faults and Spectral Information for Logic BIST," in *Proceedings of International Conferance on Computer-Aided Design*, pp. 294–298, 2002.

[7] X. Chen and M. Hsiao, "Testing Embedded Sequential Cores in Parallel Using Spectrum-Based BIST," *IEEE Transactions on Computers*, vol. 55, pp. 150–162, Feburary 2006.

[8] X. Chen and M. Hsiao, "Characteristic States and Cooperative Game Based Search for Efficient Sequential ATPG and Design Validation," in *Proceedings of International Test Conference*, 2006.

[9] X. Chen and M. Hsiao, “Energy-Efficient Logic BIST Based on State Correlation Analysis,” in *Proceedings of VLSI Test Symposium*, pp. 267–272, 2003.

[10] D. Saab, Y. Saab, and J. Abraham, “CRIS: A Test Cultivation Program for VLSI Circuits,” in *Proceedings of International Conferance on Computer-Aided Design*, pp. 216–219, 1992.

[11] I. Pomeranz and S. Reddy, “LOCSTEP: A Logic-Simulation-Based Test Generation Procedure,” *IEEE Transactions on Computer-Aided Design*, vol. 16, pp. 544–554, May 1997.

[12] S. Sheng, A. Jain, M. Hsiao, and V. Agrawal, “Correlation Analysis for Compacted Test Vectors and the Use of Correlated Vectors for Test Generation,” in *Internatial Test Synthesis Workshop*, pp. 8–15, 2000.

[13] Q. Wu and M. Hsiao, “Efficient Sequential ATPG Based on Partitioned-Finite-State Traversal,” in *Proceedings of International Test Conference*, pp. 281–289, 2003.

[14] Q. Wu and M. Hsiao, “State Variable Extraction to Reduce Problem Complexity for ATPG and Design Validation,” in *Proceedings of International Test Conference*, pp. 820–829, 2004.

[15] H. Wunderlich, “Multiple Distribution for Biased Random Test Patterns,” in *Proceedings of International Test Conference*, pp. 236–264, 1988.

[16] F. Muradali, T. Nishada, and T. Shimizu, “Structure and Technique for Pseudo Random-based Testing of Sequential Circuits,” *Journal of Electronic Testing: Theory and Applications*, vol. 6, pp. 107–115, Feburary 1995.

[17] M. Alshaibi and C. Kime, “Fixed-biased Pseudorandom Built in Self Test for Random Pattern Resistant Circuits,” in *Proceedings of International Test Conference*, pp. 929–938, 1994.

[18] M. Bershteyn, “Calculation of Multiple Sets of Weights for Weighted Random Testing,” in *Proceedings of International Test Conference*, pp. 1031–1040, 1993.

[19] D. Kagaris and S. Tragoudas, "Generating Deterministic Unordered Test Patterns with Counter," in *Proceedings of VLSI Test Symposium*, pp. 374–379, 1996.

[20] M. Chatterjee and D. K. Pradhan, "A Novel Pattern Generator for Near-Perfect Fault Coverage," in *Proceedings of VLSI Test Symposium*, pp. 417–425, 1995.

[21] B. Koenemann, "LFSR-Coded Test Patterns for Scan Designs," in *Proceedings of European Test Conference*, pp. 237–242, 1991.

[22] S. Venkataraman, J. Rajski, S. Hellebrand, and S. Tarnick, "An Efficient BIST Scheme Based on Reseeding of Multiple Polynomial Linear Feedback Shift Registers," in *Proceedings of International Conferance on Computer-Aided Design*, pp. 572–577, 1993.

[23] C. Fagot, O. Gascuel, P. Girard, and C. Landrault, "On Calculating Efficient LFSR Seeds for Built-In Self-Test," in *Proceedings of European Test Workshop*, pp. 7–14, 1999.

[24] S. Gerstendorfer and H. Wunderlich, "Minimized Power Consumption for Scan-Based BIST," in *Proceedings of International Test Conference*, pp. 77–84, 1999.

[25] V. Dabholkar, S. Chakravarty, I. Pomeranz, and S. Reddy, "Techniques for Minimizing Power Dissipation in Scan and Combinational Circuits During Test Application," *IEEE Transactions on Computer-Aided Design*, vol. 17, pp. 1325–1333, December 1998.

[26] S. Chakravarty and V. Dabholkar, "Two Techniques for Minimizing Power Dissipation in Scan Circuits During Test Application," in *Proceedings of Asian Test Symposium*, pp. 324–329, 1994.

[27] S. Wang and S. Gupta, "DS-LFSR: A New BIST TPG for Low Heat Dissipation," in *Proceedings of International Test Conference*, pp. 848–857, 1997.

[28] O. Sinanoglu, I. Bayraktaroglu, and A. Orailoglu, "Test Power Reduction Through Minimization of Scan Chain Transitions," in *Proceedings of VLSI Test Symposium*, pp. 166–171, 2002.

[29] P. Girard, L. Guiller, C. Landrault, and S. Pravossoudovitch, "A Test Vector Inhibiting Technique for Low Energy BIST Design," in *Proceedings of VLSI Test Symposium*, pp. 407–412, 1999.

[30] R. Sankaralingam and N. Touba, "Reducing Test Power During Test Using Programmable Scan Chain Disable," in *Electronic Design, Test and Applications*, pp. 159–163, 2002.

[31] L. Whetsel, "Adapting Scan Architectures for Low Power Operation," in *Proceedings of International Test Conference*, pp. 863–872, 2000.

[32] S. Wang and S. Gupta, "LT-RTPG: A New Test-Per-Scan BIST TPG for Low Heat Dissipation," in *Proceedings of International Test Conference*, pp. 85–94, 1999.

[33] S. Wang, "Generation of Low Power Dissipation and High Fault Coverage Patterns for Scan-Based BIST," in *Proceedings of International Test Conference*, pp. 834–843, 2002.

[34] R. Sankaralingam and N. Touba, "Controlling Peak Power During Scan Testing," in *Proceedings of VLSI Test Symposium*, pp. 153–159, 2002.

[35] F. Corno, M. Rebaudengo, M. Reorda, and M. Violante, "On Reducing the Peak Power Consumption of Test Sequences," in *Proceedings of European Conference on Circuit Theory & Design*, pp. 247–250, 1999.

[36] P. Girard, L. Guiller, C. Landrault, and S. Pravossoudovitch, "Circuit Partitioning for Low Power BIST Design with Minimized Peak Power Consumption," in *Proceedings of Asian Test Symposium*, pp. 89–94, 1999.

[37] P. Girard, "Survey of Low-Power Testing of VLSI Circuits," *IEEE Transactions on Design & Test of Computers*, vol. 19, pp. 82–92, May 2002.

[38] M.-W. Corporation, "Merriam-Webster Online," in *http://www.m-w.com*.

[39] K. Radecka, J. Rajski, and J. Tyszer, "Arithmetic Built-In-Self-Test for DSP Cores," *IEEE Transactions on Computer-Aided Design*, vol. 16, pp. 1358–1369, November 1997.

[40] S. Hellebrand and H. Wunderlich, "Mixed-mode BIST Using Embedded Processors," in *Proceedings of International Test Conference*, pp. 195–204, 1996.

[41] R. Dorsch and H. Wunderlich, "Accumulator Based Deterministic BIST," in *Proceedings of International Test Conference*, pp. 412–421, 1998.

[42] L. Chen and S. Dey, "DEFUSE: A Deterministic Functional Self-Test Methodology for Processors," in *Proceedings of VLSI Test Symposium*, pp. 255–262, 2000.

[43] A. Giani, S. Sheng, M. Hsiao, and V. Agrawal, "Novel Spectral Methods for Built-In Self-Test in a System-On-A-Chip Environment," in *Proceedings of VLSI Test Symposium*, pp. 163–168, 2001.

[44] A. Giani, S. Sheng, M. Hsiao, and V. Agrawal, "Efficient Spectral Techniques for Sequential ATPG," in *Proceedings of Design, Automation & Test in Europe*, pp. 204–208, 2001.

[45] C. Fagot, P. Girard, and C. Landrault, "On Using Machine Learning for Logic BIST," in *Proceedings of International Test Conference*, pp. 338–346, 1997.

[46] A. Raghunathan and S. Chakradhar, "Acceleration Techniques for Dynamic Vector Compaction," in *Proceedings of International Conferance on Computer-Aided Design*, pp. 310–317, 1995.

[47] F. Brglez, D. Bryan, and K. Kozminski, "Combinational Profiles of Sequential Benchmark Circuits," in *Proceedings of International Symposium on Circuits and Systems*, pp. 1929–1934, 1989.

[48] S. Davidson and Panelists, "ITC '99 Benchmark Circuits-Preliminary Results," in *Proceedings of International Test Conference*, p. 1125, 1999.

[49] M. Hsiao, E. Rudnick, and J. Patel, "Dynamic State Traversal for Sequential Circuit Test Generation," *ACM Transactions on Design Automation of Electronic Systems*, vol. 5, pp. 548–565, July 2000.

[50] E. Larsson, Z. Peng, and G. Carlsson, "The Design and Optimization of SOC Test Solutions," in *Proceedings of International Conferance on Computer-Aided Design*, pp. 523–530, 2001.

[51] Z. Ebadi and A. Ivanov, "Time Domain Multiplexed TAM: Implementation and Comparison," in *Proceedings of Design, Automation & Test in Europe*, pp. 10732–10737, 2003.

[52] I. Ghosh, N. Jha, and S. Dey, "A Low Overhead Design for Testability and Test Generation Technique for Core-Based Systems," in *Proceedings of International Test Conference*, pp. 50–59, 1999.

[53] L. Whetsel, "An IEEE 1149.1 Based Test Access Architecture for Ic with Embedded Cores," in *Proceedings of International Test Conference*, pp. 69–78, 1997.

[54] N. Touba and B. Pouya, "Testing Embedded Cores Using Partial Isolation Rings," in *Proceedings of International Test Conference*, pp. 10–16, 1997.

[55] M. Nahvi and A. Ivanov, "A Packet Switching Communication-Based Test Access Mechanism for System Chips," in *Proceedings of European Test Workshop*, pp. 81–86, 2001.

[56] E. Marinissen, "A Structures and Scalable Mechanism for Test Access to Embedded Reusable Cores," in *Proceedings of International Test Conference*, pp. 284–293, 1998.

[57] L. Whetsel, “Addressable Test Ports: an Approach to Testing Embedded Cores,” in *Proceedings of International Test Conference*, pp. 1055–1064, 1999.

[58] T. Niermann and J. Patel, “HITEC: A Test Generation Package for Sequential Circuits,” in *Proceedings of European Design Automation Conferance*, pp. 214–218, 1991.

[59] R. Guo, S. Reddy, and I. Pomeranz, “PROPTEST: a Property Based Test Pattern Generator for Sequential Circuits Using Test Compaction,” in *Proceedings of Design Automation Conferance*, pp. 653–659, 1999.

[60] Q. Wu and M. Hsiao, “Efficient ATPG for Design Validation Based on Partitioned State Histories,” in *Proceedings of VLSI Test Symposium*, pp. 389–394, 2004.

[61] M. Braun, S. Fine, and A. Ziv, “Enhancing the Efficiency of Bayesian Network Based Coverage Directed Test Generation,” in *Proceedings of Design Automation Conferance*, pp. 286–291, 2003.

[62] A. MacKenzie and S. Wicker, “Game Theory and the Design of Self-Configuring, Adaptive Wireless Networks,” *IEEE Transactions Communications Magazine*, vol. 39, pp. 126–131, November 2001.

[63] E. Baccarelli, M. Biagi, C. Pelizzoni, and R. Cusani, “A Distributed Power Management Game for Multi-Antenna Multiple-Access for “Ad-Hoc” Networks,” in *Proceedings of Hawaii International Conference on System Sciences*, pp. 72–80, 2005.

[64] L. Zhao and M. Muller, “Game-SAT: a Preliminary Report,” in *Proceedings of International Conference on Theory and Applications of Satisfiability Testing*, p. @, 2004.

[65] F. Zagare, *Game Theory : Concepts and Applications*. 101 Productions, 1984.

[66] M. Hsiao, “Peak Power Estimation Using Genetic Spot Optimization for Large VLSI Circuits,” in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, pp. 175–180, 1999.

[67] U. of Stanford, “Murϕ Description Language and Verifier,” in *http://verify.stanford.edu/cgi-bin/wrap/dill/Murphi*.

[68] V. Boppana, S. Rajan, K. Takayama, and M. Fujita, “Model checking based on sequential atpg,” in *Proceedings of the 11th International Conference on Computer Aided Verification*, pp. 418–430, 1999.

[69] S. Sheng, K. Takayama, and M. Hsiao, “Effective Safety Property Checking Using Simulation-Based Sequential ATPG,” in *Proceedings of Design Automation Conferance*, pp. 813–818, 2002.

[70] C. Yang and D. Dill, “Validation with Guided Search of the State Space,” in *Proceedings of Design Automation Conferance*, pp. 599–604, 1998.

[71] S. Sheng and M. Hsiao, “Efficient Sequential Test Generation Based on Logic Simulation,” *IEEE Transactions on Design & Test of Computers*, vol. 19, pp. 56–64, September 2002.

[72] K. Ravi and F. Somenzi, “High Density Reachability Analysis,” in *Proceedings of International Conferance on Computer-Aided Design*, pp. 154–158, 1995.

[73] H. Cho, G. Hachtel, E. Macii, B. Plessier, and F. Somenzi, “Algorithms for Approximate FSM Traversal,” in *Proceedings of Design Automation Conferance*, pp. 25–30, 1993.

[74] S. Govindaraju and D. Dill, “Verification by Approximate Forward and Backward Reachability,” in *Proceedings of International Conferance on Computer-Aided Design*, pp. 366–400, 1998.

[75] M. Ganai, A. Aziz, and A. Kuehlmann, "Enhancing Simulation with BDDs and ATPG," in *Proceedings of Design Automation Conferance*, pp. 385–390, 1999.

[76] U. of Colorado, "VIS: A system for Verification and Synthesis," in *http://vlsi.colorado.edu/ vis/*.

[77] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *Proceedings of Design Automation Conferance*, pp. 530–535, 2001.

[78] I. Pomeranz and S. Reddy, "On Static Compaction of Test Sequences for Synchronous Sequential Circuits," in *Proceedings of Design Automation Conferance*, pp. 215–220, 1998.