Automated Test Grading and Pattern Selection for Small-Delay Defects

by

Mahmut Yilmaz

Department of Electrical and Computer Engineering
Duke University

Date: ____________________________

Approved:

Prof. Krishnendu Chakrabarty, Chair

Prof. Mohammad Tehranipoor

Prof. Hisham Massoud

Prof. Kishor S. Trivedi

Prof. Miguel A. Medina, Jr.

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

2009
ABSTRACT

Automated Test Grading and Pattern Selection for Small-Delay Defects

by

Mahmut Yilmaz

Department of Electrical and Computer Engineering
Duke University

Date: __________________________

Approved:

Prof. Krishnendu Chakrabarty, Chair

Prof. Mohammad Tehranipoor

Prof. Hisham Massoud

Prof. Kishor S. Trivedi

Prof. Miguel A. Medina, Jr.

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

2009
Abstract

Timing-related defects are becoming increasingly important in nanometer-technology integrated circuits (ICs). Small delay variations induced by crosstalk, process variations, power-supply noise, as well as resistive opens and shorts can potentially cause timing failures in a design, thereby leading to quality and reliability concerns. All these effects are noticeable in today’s technologies and they are likely to become more prominent in the next-generation process technologies [1].

The detection of small-delay defects (SDDs) is difficult because of the small size of the introduced delay. Although the delay introduced by each SDD is small, the overall impact can be significant if the target path is critical, has low slack, or includes many SDDs. The overall delay of the path may become larger than the clock period, causing circuit failure or temporarily incorrect results. As a result, the detection of SDDs typically requires fault excitation through least-slack paths. However, widely-used automatic test-pattern generation (ATPG) techniques are not effective at exciting small delay defects. On the other hand, the usage of commercially available timing-aware tools is expensive in terms of pattern count inflation and very high test-generation times. Furthermore, these tools do not target real physical defects.

SDDs are induced not only by physical defects, but also by run-time variations such as crosstalk and power-supply noise. These variations are ignored by today’s commercial ATPG tools. As a result, new methods are required for comprehensive coverage of SDDs.

Test data volume and test application time are also major concerns for large
industrial circuits. In recent years, many compression techniques have been proposed and evaluated using industrial designs. However, these methods do not target sequence- or timing-dependent failures while compressing the test patterns. Since timing-related failures in high-performance integrated circuits are now increasingly dominated by SDDs, it is necessary to develop timing-aware compression techniques.

This thesis addresses the problem of selecting the most effective test patterns for detecting SDDs. A new gate and interconnect delay-defect probability measure is defined to model delay variations for nanometer technologies. The proposed technique intelligently selects the best set of patterns for SDD detection from a large pattern set generated using timing-unaware ATPG. It offers significantly lower computational complexity and it excites a larger number of long paths compared to previously proposed timing-aware ATPG methods. It is shown that, for the same pattern count, the selected patterns are more effective than timing-aware ATPG for detecting small delay defects caused by resistive shorts, resistive opens, process variations, and crosstalk. The proposed technique also serves as the basis for an efficient SDD-aware test compression scheme. The effectiveness of the proposed technique is highlighted for industrial circuits.

In summary, this research is targeted at the testing of SDDs caused by various underlying reasons. The proposed techniques are expected to generate high-quality and compact test patterns for various types of defects in nanometer ICs. The results of this research are expected to provide low-cost and effective test methods for nanometer devices, and they will lead to higher shipped-product quality.
Contents

Abstract iv

List of Tables xi

List of Figures xiv

List of Abbreviations xxv

Acknowledgements xxvii

1 Introduction 1

1.1 Background ................................ 1

1.1.1 Fault Models ................................ 1

1.1.2 Fault Simulation ........................... 9

1.1.3 Automatic Test Pattern Generation ............ 12

1.1.4 Design for Testability ....................... 16

1.1.5 Test Data Compression ...................... 31

1.2 Motivation for Thesis Research ................... 36

1.2.1 Difficulty of Detecting SDDs .................. 37

1.2.2 The Need for Layout-aware ATPG ............... 38

1.2.3 Limitations of Existing Commercial Timing-aware ATPG Tools 40

1.2.4 The Need for Reducing Test Data Volume .......... 41

1.2.5 Summary of Research Needs ................... 43

1.3 Overview of Related Work ........................ 44
2 Test-Pattern Grading and Pattern Selection

2.1 Probabilistic Delay-Fault Model and Output Deviations for SDDs

2.1.1 Gate-Delay Defect Probabilities

2.1.2 Propagation of Signal-Transition Probabilities

2.1.3 Implementation of Algorithm for Propagating Signal-Transition Probabilities

2.1.4 Pattern-Selection Method

2.2 Experimental Results

2.2.1 Finding Gate-Delay Defect Probabilities

2.2.2 Benchmarks

2.2.3 Experimental Setup

2.2.4 Dynamic Timing Simulation

2.2.5 Correlation Between Output Deviations and Path Lengths

2.2.6 Pattern-Selection Results

2.3 Summary

3 Layout-Aware Output Deviations

3.1 Interconnect-Aware Output Deviations for Modeling Wire-Delay Variations

3.2 Experimental Results

3.2.1 Experimental Set-up

3.2.2 Generating DDPMs for Gate Instances and Interconnects
6 Industrial Applications

6.1 Background: Timing-aware FastScan .......................... 183
   6.1.1 Algorithm ........................................... 183
   6.1.2 Optimization Options ............................... 184
   6.1.3 Delay Test Quality Metrics .......................... 186

6.2 Experimental Results: Timing-aware ATPG vs n-detect TDF ATPG 186
   6.2.1 Experimental Setup .................................. 187
   6.2.2 Benchmarks ........................................... 187
   6.2.3 Simulation Results ................................... 188
   6.2.4 Summary of Timing-aware ATPG Evaluation ......... 196

6.3 Experimental Results: Application of Deviation-Driven Pattern Selection Method .......................... 197
   6.3.1 Experimental Setup .................................. 198
   6.3.2 Benchmarks ........................................... 198
   6.3.3 Simulation Results ................................... 199

6.4 Summary .................................................. 202

7 Conclusions and Future Work

7.1 Thesis Contributions ........................................ 209
7.2 Future Work ................................................. 210
List of Tables

2.1 Example DDPM for a 2-input OR gate ........................................ 50
2.2 Example DDPM for AND, XOR, INV ............................................ 58
2.3 Benchmark statistics ................................................................. 67
2.4 Kendall’s coefficients for evaluating the correlation of path lengths to output deviations ......................................................... 70
2.5 ATPG tool CPU time (in seconds) used to generate initial pattern sets 75
2.6 Number of test patterns generated by the commercial ATPG tool for various $n$-detect TDF and timing-aware ATPG (ta) ................... 75
2.7 Number of test patterns (one-third of base set) selected by the dynamic-timing simulation based scheme (dts). ............................ 75
2.8 Number of test patterns selected by the deviation-based scheme (dev) when $D_{LIMIT} = 0.6$ and $D_{LIMIT} = 0.8$. ................................. 76
3.1 Example DDPM for net $\xi$ of Fig. 3.1 ........................................... 114
3.2 Example DDPM for each port-to-port path on net $\xi$ of Fig. 3.1 .... 114
3.3 Example DDPM for distributed delay model on net $\xi$ of Fig. 3.1 ... 114
3.4 Number of selected patterns $S$ .................................................... 125
3.5 Number of top-off patterns added to the selected patterns to achieve the same TDF coverage as timing-aware ............................ 125
3.6 Kendall’s coefficients for evaluating the correlation of path lengths to output deviations ............................................................... 126
3.7 Distribution of local and global wires on long paths ................. 127

3.8 Number of delay faults detected using selected patterns and Timing-aware ATPG patterns (a) with delay variations only on gates (b) with delay variations on gates and interconnects (c) with RORS. ............. 129

3.9 The percentage increase in the number of long paths activated by the lumped-delay model relative to the wire-delay-oblivious model. ........ 131

3.10 The percentage of different patterns selected at 180nm and 45nm process technologies. ................................................. 132

4.1 Details of Experimental Benchmarks. ................................. 155

4.2 Number of sensitized long paths for different pattern set. .......... 158

4.3 Number of sensitized SDDs for different pattern set. ............... 158

4.4 Number of patterns for different pattern set. ...................... 159

4.5 Long path threshold impact on pattern selection results for tv80 benchmark. ............................................................... 160

5.1 Benchmark statistics ....................................................... 168

5.2 The comparison of number of excited distinct long paths for a long path limit of 70% of the clock period. ............................. 169

5.3 The comparison of number of excited distinct long paths for a long path limit of 80% of the clock period. ............................. 169

5.4 The comparison of number of excited distinct long paths for a long path limit of 90% of the clock period. ...................... 170

5.5 Benchmark run-time statistics ........................................... 174

5.6 Fault injection results: Number of detected faults ............... 174
6.1 Fault counts for industrial circuit blocks ................. 187

6.2 Normalized test cost statistics for $n$-detect TDF and TA ATPG: Pattern count comparison, normalized by $n$-1. ........................................ 192

6.3 Normalized test cost statistics for $n$-detect TDF and TA ATPG: CPU time comparison, normalized by $n$-1. .......................... 193

6.4 Normalized test cost statistics for $n$-detect TDF and TA ATPG: Memory usage comparison, normalized by $n$-1. ............................. 194

6.5 Gate counts for industrial circuit blocks ......................... 198

A.1 DDPM for a 2-input OR gate instance ......................... 213

A.2 DDPM for a 2-input AND gate instance ....................... 213

A.3 DDPM for a 2-input XOR gate instance ....................... 214

A.4 DDPM for three different INVERTER gate instance ........... 214
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Stuck-at fault example</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Delay-defect timing</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>Generic hardware model and clock timing</td>
<td>5</td>
</tr>
<tr>
<td>1.4</td>
<td>Intra-layer bridging fault</td>
<td>7</td>
</tr>
<tr>
<td>1.5</td>
<td>Inter-layer bridging fault</td>
<td>7</td>
</tr>
<tr>
<td>1.6</td>
<td>Bridging fault examples</td>
<td>8</td>
</tr>
<tr>
<td>1.7</td>
<td>Fault collapsing example: (a) Initial fault list; (b) Fault equivalence; (c) Fault dominance</td>
<td>11</td>
</tr>
<tr>
<td>1.8</td>
<td>A simple example of redundant fault. Stuck-at-1 fault is redundant on net B</td>
<td>13</td>
</tr>
<tr>
<td>1.9</td>
<td>A basic ATPG flow</td>
<td>15</td>
</tr>
<tr>
<td>1.10</td>
<td>A scan chain example</td>
<td>17</td>
</tr>
<tr>
<td>1.11</td>
<td>Scan test is initialized by forcing scan enable signal</td>
<td>24</td>
</tr>
<tr>
<td>1.12</td>
<td>Shift-in operation started</td>
<td>25</td>
</tr>
<tr>
<td>1.13</td>
<td>The test vector is completely shifted-in</td>
<td>25</td>
</tr>
<tr>
<td>1.14</td>
<td>PIs are forced and combinational blocks generated outputs</td>
<td>26</td>
</tr>
<tr>
<td>1.15</td>
<td>System clock is toggled</td>
<td>26</td>
</tr>
<tr>
<td>1.16</td>
<td>Combinational block outputs are captured in scan flip-flops. The next test vector is ready to be shifted-in</td>
<td>27</td>
</tr>
</tbody>
</table>
1.17 While captured responses of the previous vector is shifted-out, the new test vector is shifted-in. .......................... 27

1.18 The responses of the previous vector is completely shifted-out, the new test vector is completely shifted-in. .................. 28

1.19 Initial steps for LOC delay test generates the same responses as stuck-at test. ........................................ 28

1.20 Second test inputs are generated by a second PI force and a clock toggle. 29

1.21 Timing diagram for LOC delay test. .............................. 29

1.22 LOS delay test starts like the LOC delay test. ...................... 30

1.23 Second test inputs are generated by shifting scan chain values. .... 30

1.24 Timing diagram for LOS delay test. .............................. 31

1.25 A high level view of the LFSR-reseeding-based test data compression scheme. ........................................ 33

1.26 Transistor gate length over the years [2]. ....................... 36

1.27 Delay-defect distribution [3]. ...................................... 38

1.28 CPU time of timing-aware ATPG relative to traditional TDF ATPG for the IWLS 2005 benchmark circuits. .................. 42

1.29 CPU time and pattern count of timing-aware ATPG relative to traditional TDF ATPG for sub-blocks of an industrial microprocessor circuit. ........................................ 43

2.1 An example Gaussian delay distribution curve to illustrate $T_{CRT}$ and DDP. The shaded area corresponds to the defect region (DDP). . . . 51
2.2 An example non-Gaussian delay distribution curve to illustrate $T_{CRT}$ and DDP. The shaded area corresponds to the defect region (DDP). 52

2.3 Examples to illustrate the propagation rules. 55

2.4 An example to illustrate the propagation of signal-transition probabilities through the gates of a logic circuit. 57

2.5 Signal transition probability propagation algorithm for calculating output deviations. 61

2.6 Deviation-computation algorithm for pattern selection. 63

2.7 Pattern selection algorithm. 64

2.8 Change in DDPM values as a function of the number of MC simulations, relative to the base case of 50 MC simulations. (a) OR gate, (b) AND gate, (c) INV, BUF, and XOR gates. 74

2.9 Comparison of CPU time for dev and dts. The CPU times used for each step is shown in a different color. The values are normalized by the timing-aware ATPG CPU time for each benchmark. The normalization point (timing-aware ATPG CPU time) is shown as a red line. 77

2.10 The number of sensitized distinct long paths for dev and dts (LPL=90%): (a) systemcaes; (b) usb_func; (c) ac97_ctrl; (d) aes_core; (e) pci_bridge32; (f) wb_conmax; (g) ethernet. The number of sensitized distinct long paths for the trimmed timing-aware ATPG patterns is shown as a horizontal line. 78

2.11 The number of sensitized distinct long paths by dev and dts for benchmark systemcaes compared to full base pattern sets (LPL=90%): (a) $D_{LIMIT} = 0.8$; (b) $D_{LIMIT} = 0.3$; (c) Pattern count comparison table. 79
2.12 The number of sensitized distinct long paths by `dev` and `dts` for benchmark `usb_funct` compared to full base pattern sets (LPL=90%): (a) $D_{LIMIT} = 0.8$; (b) $D_{LIMIT} = 0.3$; (c) Pattern count comparison table. 80

2.13 The number of sensitized distinct long paths by `dev` and `dts` for benchmark `ac97_ctl` compared to full base pattern sets (LPL=90%): (a) $D_{LIMIT} = 0.8$; (b) $D_{LIMIT} = 0.3$; (c) Pattern count comparison table. 81

2.14 The number of sensitized distinct long paths by `dev` and `dts` for benchmark `aes_core` compared to full base pattern sets (LPL=90%): (a) $D_{LIMIT} = 0.8$; (b) $D_{LIMIT} = 0.3$; (c) Pattern count comparison table. 82

2.15 The number of sensitized distinct long paths by `dev` and `dts` for benchmark `pci_bridge32` compared to full base pattern sets (LPL=90%): (a) $D_{LIMIT} = 0.8$; (b) $D_{LIMIT} = 0.3$; (c) Pattern count comparison table. 83

2.16 The number of sensitized distinct long paths by `dev` and `dts` for benchmark `wb_conmax` compared to full base pattern sets (LPL=90%): (a) $D_{LIMIT} = 0.8$; (b) $D_{LIMIT} = 0.3$; (c) Pattern count comparison table. 84

2.17 The number of sensitized distinct long paths by `dev` and `dts` for benchmark `ethernet` compared to full base pattern sets (LPL=90%): (a) $D_{LIMIT} = 0.8$; (b) $D_{LIMIT} = 0.3$; (c) Pattern count comparison table. 85

2.18 The distribution of sensitized distinct paths over 80% of the clock period for base 8-detect patterns, `dts`-based selection, and `dev`-based selection: (a) `systemcaes`; (b) `usb_funct`; (c) `ac97_ctl`; (d) `aes_core`. 87

2.19 The distribution of sensitized distinct paths (LPL=80%) for base 8-detect patterns, `dts`-based selection, and `dev`-based selection: a) `pci_bridge32`; (b) `wb_conmax`; (c) `ethernet`. 88

2.20 The distribution of injected delay defects. 89
2.21 The number of defects detected by the selected patterns for dev, dts, the trimmed timing-aware ATPG patterns (ta_trimmed), and full set of timing-aware ATPG patterns (ta) (ta is a super-set of all selected patterns): (a) systemcaes; (b) usb_funct; (c) ac97_ctrl; (d) aes_core; (e) pci_bridge32; (f) wb_conmax; (g) ethernet. ................................. 91

2.22 The fault coverage ramp-up using the selected patterns of dev and dts, and the base timing-aware ATPG patterns: systemcaes. ....... 92

2.23 The fault coverage ramp-up using the selected patterns of dev and dts, and the base timing-aware ATPG patterns: (a) usb_funct; (b) ac97_ctrl. ................................................................. 93

2.24 The fault coverage ramp-up using the selected patterns of dev and dts, and the base timing-aware ATPG patterns: (a) aes_core; (b) pci_bridge32. ................................................................. 94

2.25 The fault coverage ramp-up using the selected patterns of dev and dts, and the base timing-aware ATPG patterns: (a) wb_conmax; (b) ethernet. ................................................................. 95

2.26 The long-path coverage ramp-up using the selected patterns of dev and dts, and the full base timing-aware ATPG patterns (LPL=90%): (a) systemcaes; (b) usb_funct. ................................................................. 96

2.27 The long-path coverage ramp-up using the selected patterns of dev and dts, and the full base timing-aware ATPG patterns (LPL=90%): (a) ac97_ctrl; (b) aes_core. ................................................................. 97

2.28 The long-path coverage ramp-up using the selected patterns of dev and dts, and the full base timing-aware ATPG patterns (LPL=90%): (a) pci_bridge32; (b) wb_conmax. ................................................................. 98

2.29 The long-path coverage ramp-up using the selected patterns of dev and dts, and the full base timing-aware ATPG patterns (LPL=90%): ethernet. ................................................................. 99
2.30 The fault coverage ramp-up using the selected patterns of dev and dts, and the full base timing-aware ATPG patterns: systemcases.

2.31 The fault coverage ramp-up using the selected patterns of dev and dts, and the full base timing-aware ATPG patterns: (a) usb_funct; (b) ac97_ctrl.

2.32 The fault coverage ramp-up using the selected patterns of dev and dts, and the full base timing-aware ATPG patterns: (a) aes_core; (b) pci_bridge32.

2.33 The fault coverage ramp-up using the selected patterns of dev and dts, and the full base timing-aware ATPG patterns: (a) wb_conmax; (b) ethernet.

2.34 The effect of DDPM perturbations to the selected pattern quality in terms of sensitized long paths for benchmark systemcases. The perturbation limits are given in parentheses. (a) LPL=90%; (b) LPL=70%

2.35 The effect of DDPM perturbations to the selected pattern quality in terms of sensitized long paths for benchmark usb_funct. The perturbation limits are given in parentheses. (a) LPL=90%; (b) LPL=70%

2.36 The effect of DDPM perturbations to the selected pattern quality in terms of sensitized long paths for benchmark ac97_ctrl. The perturbation limits are given in parentheses. (a) LPL=90%; (b) LPL=70%

2.37 The effect of DDPM perturbations to the selected pattern quality in terms of sensitized long paths for benchmark aes_core. The perturbation limits are given in parentheses. (a) LPL=90%; (b) LPL=70%

2.38 The effect of DDPM perturbations to the selected pattern quality in terms of sensitized long paths for benchmark pci_bridge32. The perturbation limits are given in parentheses. (a) LPL=90%; (b) LPL=70%
2.39 The effect of DDPM perturbations to the selected pattern quality in terms of sensitized long paths for benchmark wb_conmax. The perturbation limits are given in parentheses. (a) LPL=90%; (b) LPL=70%  108

2.40 The effect of DDPM perturbations to the selected pattern quality in terms of sensitized long paths for benchmark ethernet. The perturbation limits are given in parentheses. (a) LPL=90%; (b) LPL=70%  109

3.1 Different wires for net $\xi$. 112

3.2 The overall flow for layout-aware pattern selection. 119

3.3 The comparison of CPU run-time for the proposed method and timing-aware ATPG, using various values of $n$ on IWLS benchmarks. 121

3.4 The distribution of the run-time over different steps of the proposed method, for $n=3$. 122

3.5 The distribution of the run-time over different steps of the proposed method, for $n=5$. 122

3.6 The distribution of the run-time over different steps of the proposed method, for $n=8$. 123

3.7 The distribution of the run-time over different steps of the proposed method, for $n=10$. 123

3.8 The normalized number of excited distinct long paths by selected patterns using output-deviation-based method (relative to timing-aware ATPG patterns). 126

3.9 The number of detected delay faults for the proposed method and timing-aware ATPG for benchmark tv80. 130

3.10 The comparison of CPU run-time for the proposed method and timing-aware ATPG, using various values of $n$ on IWLS benchmarks, when the pin-to-pin lumped wire delay model is used. 132

xx
4.1 Monte-Carlo simulation results on NAND3X1 gate. ................. 137
4.2 Gate and interconnect *pdf* propagation through the target path. . 138
4.3 A small circuit used to highlight the crosstalk problem. ............ 140
4.4 Layout representation of the circuit in Fig. 4.3. Crosstalk effects are
shows with arrows. ........................................ 141
4.5 Impact of aggressor arrival time on victim propagation delay when
victim and aggressor nets have (a) same transition direction and (b)
opposite transition direction. .................................. 143
4.6 Propagation delays when the coupling capacitance between victim and
aggressor net increases assuming that both aggressor and victim nets
have same arrival times. ....................................... 144
4.7 Crosstalk impact on interconnect *pdfs*. .......................... 146
4.8 Path *pdf* and delay distribution. ............................... 149
4.9 An example of path evaluation and pattern evaluation. .............. 150
4.10 Pattern sorting algorithm. ..................................... 152
4.11 Flow diagram of pattern generation, evaluation and selection. .... 153
4.12 Correlation between the number of sensitized long paths and number
of detected SDDs for tv80. .................................... 156
4.13 Long path detection of selected patterns for tv80. .................... 157
5.1 Procedure to select LFSR seeds. ................................ 164
5.2 (a) An LFSR and phase shifter; (b) State transition matrix of the LFSR. 167
5.3 Example to illustrate the solution for system of linear equations. . . 167
5.4 The number of defects detected by the selected patterns for deviation-based (**dev**), random-fill (**rand**), and the fill-0 or fill-1 (**fill-0/1**) methods: (a) wb dma; (b) tv80; (c) mem ctrl; (d) usb funct; (e) aes core; (f) ac97 ctrl; (g) dma; (h) wb conmax; (i) pci bridge32; (j) ethernet; (k) vga_lcd ..............................

5.5 The fault coverage ramp-up using the specified patterns of deviation-based (**dev**), random-fill (**rand**), and the fill-0 or fill-1 (**fill-0/1**) methods: (a) wb dma; (b) tv80. ..............................

5.6 The fault coverage ramp-up using the specified patterns of deviation-based (**dev**), random-fill (**rand**), and the fill-0 or fill-1 (**fill-0/1**) methods: (a) mem ctrl; (b) usb funct. ..............................

5.7 The fault coverage ramp-up using the specified patterns of deviation-based (**dev**), random-fill (**rand**), and the fill-0 or fill-1 (**fill-0/1**) methods: (a) aes core; (b) ac97 ctrl. ..............................

5.8 The fault coverage ramp-up using the specified patterns of deviation-based (**dev**), random-fill (**rand**), and the fill-0 or fill-1 (**fill-0/1**) methods: (a) dma; (b) wb conmax. ..............................

5.9 The fault coverage ramp-up using the specified patterns of deviation-based (**dev**), random-fill (**rand**), and the fill-0 or fill-1 (**fill-0/1**) methods: (a) pci bridge32; (b) ethernet. ..............................

5.10 The fault coverage ramp-up using the specified patterns of deviation-based (**dev**), random-fill (**rand**), and the fill-0 or fill-1 (**fill-0/1**) methods for benchmark vga_lcd. ..............................

6.1 Normalized SDD pattern quality statistics for n-detect TDF and TA ATPG: (a) DTC; (b) SDQM. All values are normalized by the delay test coverage of TA ATPG when slack margin is set to tool default, i.e., 50% (shown as TA, $\sigma=50\%$). ..............................
6.2 Normalized CPU time usage for \( n \)-detect ATPG and TA ATPG when multiple parallel processors are used. All values are normalized by the CPU time when no slave processors are used (single processor ATPG run). ................................................... 196

6.3 Normalized number of test patterns for \( n \)-detect ATPG, timing-aware ATPG (ta), and the proposed pattern selection method (dev). Path threshold is set to 0.7 for the proposed scheme. All values are normalized by the value of \( n-1 \). .......................................................... 200

6.4 Normalized number of test patterns for \( n \)-detect ATPG, timing-aware ATPG (ta), and the proposed pattern selection method (dev). Path threshold is set to 0.8 for the proposed scheme. All values are normalized by the value of \( n-1 \). .......................................................... 201

6.5 Normalized number of test patterns for \( n \)-detect ATPG, timing-aware ATPG (ta), and the proposed pattern selection method (dev). Path threshold is set to 0.9 for the proposed scheme. All values are normalized by the value of \( n-1 \). .......................................................... 202

6.6 Normalized CPU time usage for \( n \)-detect ATPG, timing-aware ATPG (ta), and the proposed pattern grading and pattern selection method (dev). Path threshold is set to 0.8 for the proposed scheme. All values are normalized by the value of \( n-1 \). .......................................................... 203

6.7 Normalized number of sensitized distinct long paths for \( n \)-detect ATPG, timing-aware ATPG (ta), and the proposed pattern selection method (dev). Path threshold is set to 0.7 for the proposed scheme. All values are normalized by the value of \( n-1 \). .......................................................... 204

6.8 Normalized number of sensitized distinct long paths for \( n \)-detect ATPG, timing-aware ATPG (ta), and the proposed pattern selection method (dev). Path threshold is set to 0.8 for the proposed scheme. All values are normalized by the value of \( n-1 \). .......................................................... 204
6.9 Normalized number of sensitized distinct long paths for $n$-detect ATPG, timing-aware ATPG (ta), and the proposed pattern selection method (dev). Path threshold is set to 0.9 for the proposed scheme. All values are normalized by the value of $n$-1. ..................205

6.10 The long-path coverage ramp-up using the base 8-detect TDF ATPG ($n$-8(base)), timing-aware ATPG with $\sigma=25\%$ (ta,s25(base)) and $\sigma=0\%$ (ta,s0(base)), selected patterns from 8-detect TDF ATPG ($n$-8(selected)) and timing-aware ATPG (ta,s0(selected)). Long path limit is set to 80\%: (a) Circuit A; (b) Circuit B. ..................206

6.11 The long-path coverage ramp-up using the base 8-detect TDF ATPG ($n$-8(base)), timing-aware ATPG with $\sigma=25\%$ (ta,s25(base)) and $\sigma=0\%$ (ta,s0(base)), selected patterns from 8-detect TDF ATPG ($n$-8(selected)) and timing-aware ATPG (ta,s0(selected)). (a) Circuit C, long path limit is set to 80\%; (b) Circuit C, long path limit is set to 90\%. .... 207
## List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALAPTF</td>
<td>As late as possible transition</td>
</tr>
<tr>
<td>AMD</td>
<td>Advanced Micro Devices, Inc.</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application specific integrated circuit</td>
</tr>
<tr>
<td>ATE</td>
<td>Automatic test equipment</td>
</tr>
<tr>
<td>ATPG</td>
<td>Automatic test pattern generation</td>
</tr>
<tr>
<td>CMOS</td>
<td>Complementary metaloxidesemiconductor</td>
</tr>
<tr>
<td>CLT</td>
<td>Central limit theorem</td>
</tr>
<tr>
<td>CPU</td>
<td>Central processing unit</td>
</tr>
<tr>
<td>CUT</td>
<td>Circuit under test</td>
</tr>
<tr>
<td>DDP</td>
<td>Delay defect probability</td>
</tr>
<tr>
<td>DDPM</td>
<td>Delay defect probability matrix</td>
</tr>
<tr>
<td>DEF</td>
<td>Design exchange format</td>
</tr>
<tr>
<td>DFT</td>
<td>Design for test</td>
</tr>
<tr>
<td>DTC</td>
<td>Delay test coverage</td>
</tr>
<tr>
<td>DUT</td>
<td>Design under test</td>
</tr>
<tr>
<td>GB</td>
<td>Gigabyte</td>
</tr>
<tr>
<td>IC</td>
<td>Integrated circuit</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of electrical and electronics engineers</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual property</td>
</tr>
<tr>
<td>ITRS</td>
<td>International technology roadmap for semiconductors</td>
</tr>
<tr>
<td>IWLS</td>
<td>International workshop on logic and synthesis</td>
</tr>
<tr>
<td>LFSR</td>
<td>Linear-feedback shift register</td>
</tr>
<tr>
<td>LOC</td>
<td>Launch-on-capture</td>
</tr>
<tr>
<td>LOS</td>
<td>Launch-on-shift</td>
</tr>
<tr>
<td>LP</td>
<td>Long path</td>
</tr>
<tr>
<td>LPL</td>
<td>Long path limit</td>
</tr>
<tr>
<td>LSF</td>
<td>Load sharing facility</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>MC</td>
<td>Monte Carlo simulation, HSpice related</td>
</tr>
<tr>
<td>NSF</td>
<td>National Science Foundation</td>
</tr>
<tr>
<td>PDF</td>
<td>Path delay fault</td>
</tr>
<tr>
<td>pdf</td>
<td>Probability density function</td>
</tr>
<tr>
<td>PI</td>
<td>Primary input</td>
</tr>
<tr>
<td>PN</td>
<td>Processed net</td>
</tr>
<tr>
<td>PO</td>
<td>Primary output</td>
</tr>
<tr>
<td>RORS</td>
<td>Resistive open and resistive shorts</td>
</tr>
<tr>
<td>RTL</td>
<td>Register-transfer level</td>
</tr>
<tr>
<td>SCOAP</td>
<td>Sandia controllability and observability analysis program</td>
</tr>
<tr>
<td>SDD</td>
<td>Small-delay defect</td>
</tr>
<tr>
<td>SDF</td>
<td>Standard delay format</td>
</tr>
<tr>
<td>SDQL</td>
<td>Statistical delay quality level</td>
</tr>
<tr>
<td>SDQM</td>
<td>Statistical delay quality model</td>
</tr>
<tr>
<td>SE</td>
<td>Scan enable</td>
</tr>
<tr>
<td>SSTA</td>
<td>Statistical static-timing analysis</td>
</tr>
<tr>
<td>STA</td>
<td>Static-timing analysis</td>
</tr>
<tr>
<td>STP</td>
<td>Signal transition probability</td>
</tr>
<tr>
<td>STPV</td>
<td>Signal transition probability vector</td>
</tr>
<tr>
<td>TA</td>
<td>Timing-aware</td>
</tr>
<tr>
<td>TDF</td>
<td>Transition-delay fault</td>
</tr>
<tr>
<td>VDD</td>
<td>Power supply voltage</td>
</tr>
<tr>
<td>VDSM</td>
<td>Very deep sub-micron</td>
</tr>
<tr>
<td>VLSI</td>
<td>Very large scale integration</td>
</tr>
</tbody>
</table>
Acknowledgements

This work would not have been possible without the support and supervision of my advisor, Dr. Krishnendu Chakrabarty, who gave me chance to succeed, accepted me as a PhD student, and motivated me to select this topic. I am very grateful to him for the research freedom and invaluable guidance he has provided throughout my research work. Among all other things, Dr. Chakrabarty taught me the way of scientific thinking, which I will benefit for the rest of my life.

I would like to thank Dr. Mohammad Tehranipoor, who generously accepted to be my co-advisor. Being one of the best experts in the field of physical-aware and timing-aware testing, Dr. Tehranipoor provided valuable counsel and feedback in my research work.

I thank Jeff Rearick, Jeff Fitzgerald, and other colleagues at AMD for valuable discussions and for providing me access to computing resources. Working full-time and doing PhD research at the same time would not be possible without the support of my manager, Jeff Fitzgerald. I would like to thank Dr. Anuja Sehgal for serving in my preliminary exam committee and for helping me get AMD support for this research work.

I would like to thank my committee members Dr. Hisham Massoud, Dr. Kishor Trivedi, Dr. Miguel Medina, and Dr. Mohammad Tehranipoor for taking time to serve on my dissertation committee, and for providing constructive technical feedback on my work.

I would like to thank Dr. Sule Ozev and Dr. Daniel Sorin for all their support and guidance in the first years of my graduate study.
I thank people in my research group and Prof. Tehranipoor’s research group, Ke Peng, Sudarshan Bahukudumbi, Zhanglei Wang, Hongxia Fang, and Jeremy Lee for beneficial discussions and for helping me in my research work. I would like to thank dear friends, Ufuk Okuyucu, Erdem Serkan Erdogan, Ayca Erdogan, Oguz Kazanci, Oguz Degirmenci, and Erkan Acar for all their support.

I would like to thank ECE staff, specifically Samantha Morton, Autumn Wenner, Ellen Currin, and Natasha Armstrong for helping me when I needed it.

Last, but not the least, I would like to thank to my wife Natasha, my daughter Nadia, my parents and my sister, as well as my parent-in-laws for their invaluable and unconditional support.
Chapter 1

Introduction

The testing of an integrated circuit (IC) is the process of exercising the circuit with test patterns and analyzing its responses to determine whether it behaves correctly. A form of testing, referred to as validation, is carried out at design time before the chip is sent for mass production. The purpose of validation is to diagnose and correct design bugs, and to estimate performance characteristics for setting specifications [4]. Once an IC is fabricated, it is subjected to manufacturing testing to prevent the delivery of defective parts to customers. Manufacturing testing may not detect all defective parts, but the test escapes should be minimized.

In this thesis, we focus on manufacturing testing. Specific techniques and solutions are presented for increasing the test quality for today’s high-density nanometer ICs.

1.1 Background

1.1.1 Fault Models

A fault, in the most general sense, is an imperfection in a design or implementation. In IC testing, faults are used to model fabrication defects that can lead to physical failures. An error is the manifestation of a fault in the system. The purpose of manufacturing testing is to detect the faulty circuits by forcing them to manifest faults as errors, e.g., incorrect outputs.
A fault model is an abstraction of a physical defect, which can be used to determine the effect (error) of the corresponding defect at the output of the circuit under test (CUT). Testing involves the detection of faults from a given fault model. The advantage of modeling physical defects as logical faults is that the problem of fault analysis is simplified and there is no need to explicitly target poorly understood defects. A single fault model may cover many physical defects, thereby enabling IC manufacturers to test physical defects whose effect on circuit behavior is too complex to be analyzed [5]. Commonly used fault models in practice include stuck-at faults, delay faults, bridging faults, and stuck-open faults.

**Stuck-at Fault Model.** The single stuck-at fault model (Fig. 1.1) is the most commonly used fault model due to its simplicity. The assumptions of the single stuck-at fault model are listed below.

- Only one line in the circuit is faulty.
- The fault is permanent.
- The effect of the fault is as if the faulty node is tied to either $V_{DD}$ (Power Supply) (stuck-at-1) or $GND$ (Ground) (stuck-at-0).
- The function of the gate in the circuit is unaffected by the fault.

The single stuck-at fault model offers numerous advantages. It is simple to use during test generation and fault simulation, and it covers a large percentage of physical defects. Other fault models, that can be targeted to increase defect coverage, can be mapped into sequences of single stuck-at faults. There are, however, some
Figure 1.1: Stuck-at fault example: A stuck-at-1 on net $x$.

defects that cannot be targeted by the single stuck-at model. The advantages and disadvantages of this fault model are given below.

- **Advantages**
  - Can be applied at the logic level or the module level.
  - The number of faults is reasonable ($2^n$ for a circuit with $n$ nets).
  - Algorithms for automatic test-pattern generation (ATPG) are well-developed and efficient.
  - The single-stuck at fault model can cover a fairly large portion of all possible manufacturing defects in complementary metal-oxide semiconductor (CMOS) circuits.
  - Other fault models can be mapped into a series of stuck-at faults.

- **Disadvantage**
  - It does not cover all defect types for CMOS circuits, e.g., delay faults, resistive shorts, resistive opens.

**Delay-Fault Model.** As the process technology scales down to very deep sub-micron (VDSM) levels, new process elements such as copper wiring and low-$\kappa$ dielectrics have changed the relative frequency of different defect mechanisms. As a
result, more defects tend to alter the delay of the circuit instead of changing the logic function.

Despite the popularity of the stuck-at fault model, it is no longer the only relevant fault model. The delay-fault model assumes that the faulty circuit element gets the correct value but that this value arrives too late. The delay-fault model also covers many physical phenomena that are becoming more prevalent in modern microprocessors, including the effects of temperature and process variability [6, 7].

![Diagram of delay-defect timing](image)

**Figure 1.2:** Delay-defect timing.

In Figure 1.2, the upper curve shows the defect-free output response, and the bottom curve shows the response typical of a traditional stuck-at fault (stuck-at-0 in this case). Threshold voltages are marked, and the vertical dashed line shows the required arrival time at the next logic stage for the transition to be captured in a downstream flip-flop, as predicted by static timing analysis. For a delay defect, the middle curve shows that the output goes to logic 1 too late for proper operation,
resulting in a timing failure.

Since delay faults do not alter the logic function realized by a circuit and since the tests for stuck-at faults are normally applied at a slow clock rate (due to the Automatic test equipment limitation or other practical reasons), they are inadequate for detecting delay faults. Special two-pattern test vectors are required for detecting delay faults [8].

![Generic hardware model and clock timing](image)

**Figure 1.3:** Generic hardware model and clock timing.

A generic hardware model used for delay-fault testing is shown in Figure 1.3. Here the vector pair \(<V_1, V_2>\) constitutes a delay test and signals \(Clock_1\) and \(Clock_2\) are used to clock the input and output latches, respectively. At time \(t_0\), an initializing input vector \(V_1\) is applied, and the circuit is allowed to stabilize under input \(V_1\). At time \(t_1\), the propagation vector \(V_2\) is applied, and the outputs are sampled at time \(t_2\), where \((t_2 - t_2)\) is the intended time interval between the input and output clocks, called the rated clock interval \(T_C\).

The most common delay fault-models are the transition-delay fault (TDF) model
and the path-delay fault (PDF) model.

**Transition-Delay Fault Model.** There are two kinds of TDFs, namely *slow-to-rise* and *slow-to-fall*. The slow-to-rise (fall) transition fault temporarily behaves like a DC stuck-at-0 (1) fault.

The TDF coverage is a measure of how successful the delay test is in detecting large-delay faults. The TDF mainly models defects for which the delay is at a gate large enough (gross-delay defects) to cause a logical failure when the signal propagates through any path to the test observation points [8].

**Path-Delay Fault Model.** The PDF model was first proposed in [9]. This model has been extensively studied in the literature. In the PDF model, any path with a total delay exceeding the system clock interval is said to have a *path delay fault*. The PDF model addresses distributed defects that affect an entire path, thus it is superior to TDF in its modeling capacity. The major drawback of the PDF model is that it is almost impossible to excite all the paths in the CUT and the number of paths increases exponentially with the circuit size.

**Bridging Fault Model.** The bridging fault model tests for resistive shorts between two normally unconnected instance pins or net names, which could be caused by a piece of metal from the sputtering process. Studies show that bridging faults are the closest fault models to real defects [10]. Bridging defects cannot be caught by capacitive coupling and noise analysis simulations before chip tape-out, because they are caused by fabrication defects [11].
Bridging faults can be inter-layer and intra-layer. Fig. 1.4 shows an intra-layer bridging fault, and Fig. 1.5 shows an inter-layer bridging fault.

By examining the physical layout of a design and by observing the patterns of bridging defects, we make the following conclusions regarding bridging faults. The probability of bridged net pairs [12]:

- Increases as the distance the wires run in parallel on the same layer increases;
- Increases as the distance between parallel wires decreases;
- Increases as the size of the contaminant producing bridging increases;
- Increases at via locations where two metals layers are connected;
Decreases for wires that do not run in parallel;

Decreases for wires that are on different layers.

Note that bridging faults can also occur on wires running on two different layers due to oxide layer breakdown, but the probability for this to happen is low since the area of overlap is usually small.

The bridge between nets can affect each other in different ways. For instance, one of the nets may force the other one to get the opposite value, or vice versa. Bridging faults are generally modeled as wired-OR or wired-AND. However, an ATPG tool may allow more complex modeling of a bridging fault. Most of the time, a user needs to inform the ATPG tool how the interaction between the bridging net pairs occur.
Stuck-Open Fault Model. The stuck-open fault model assumes that a net or wire is permanently open, i.e., the connection is missing [13]. The detection of stuck-open faults requires the application of specific pattern sequences since this kind of faults may require multiple test-patterns to the manifestation of an error. The advantages and disadvantages of this fault model can be summarized as follows:

- Advantages
  - Covers physical defects that are not covered by single or multiple stuck-at faults.
  - Can be tested with sequences of stuck-at fault patterns.

- Disadvantages
  - Requires a larger number of tests.
  - Algorithms for ATPG and fault simulation are complex and not well-developed.
  - Requires a lower level of circuit description (transistor level and layout) for generation of the fault list.

1.1.2 Fault Simulation

The fault-free circuit is referred to as the “good circuit”. The responses of the good circuit to the test-patterns are the expected outputs. A faulty circuit includes one or more faults from the corresponding fault model. Fault simulation is the process of simulating the faulty circuit by a test-pattern and comparing the results to the
response of good circuit. If the response of the CUT changes in the presence of the fault, this fault can be detected by the applied test-pattern. Fault simulation includes the following basic stages [14]:

**Good Circuit Simulation.** Simulation of the fault-free CUT.

**Fault specification.** Defining a set of modeled faults and performing fault-collapsing.

Fault-collapsing is the process of reducing the number of faults in the fault set by removing equivalent faults. Two faults are equivalent if they cannot be distinguished by any test-pattern. On the other hand, if a fault $F_1$ is dominant to another fault $F_2$, every test-pattern detecting $F_1$ also detects $F_2$, but the inverse is not true. Fault collapsing also removes the dominated faults (but not dominant faults).

Consider the example given in Fig. 1.7. Fig. 1.7-a shows the initial list of stuck-at faults. In 1.7-b, A stuck-at-0 and B stuck-at-0 together are equivalent to Z stuck-at-0 because only the test $AB=11$ will detect them. Therefore, A stuck-at-0 and B stuck-at-0 (or Z stuck-at-0) can be removed from the fault list due to fault equivalence. A fault dominance example is shown in Fig. 1.7-c. Z stuck-at-1 can be detected by one of the following test-patterns: $AB=00, 10, 01$. However, A stuck-at-1 can only be detected by $AB=01$. Similarly, B stuck-at-1 can only be detected by $AB=10$. As a result, A stuck-at-1 and B stuck-at-1 dominate E stuck-at-1. Z stuck-at-1 can be removed from the fault list since it is a dominated fault.

**Fault insertion.** The process of injecting a subset of faults into the CUT and chang-
Figure 1.7: Fault collapsing example: (a) Initial fault list; (b) Fault equivalence; (c) Fault dominance.

ing the circuit data structure to indicate the presence of injected faults. For instance, in Fig. 1.7, if we want to insert a stuck-at-1 fault at node A, we should tie node A to VDD so that node A has the constant 1 value independent of applied test-pattern.

Fault-effect generation and propagation. The fault injected data structures are used to generate the fault-effects and propagate them. Consider the same example in Fig. 1.7 and assume that A stuck-at-1 fault is injected. The test-pattern AB=01 can be used to generate the fault-effect of A stuck-at-1. The expected signal value, for good circuit, at net Z is 0. However, with the modified data structure, net Z will have the faulty value of 1. In the next step, the effect of this fault should be propagated to the primary outputs or test observation points.

Fault detection and discarding. If the effects of an inserted fault can be prop-
agated to primary outputs, this fault is marked as detected. It is possible to set detection limits for each fault. If the limit is set to $n$, a fault is discarded after being detected for $n$ times ($n$-detect test). Usually, $n$ is set to 1 unless it is changed by the user.

### 1.1.3 Automatic Test Pattern Generation

Automatic test pattern generation is the process of automatically generating a set of test-patterns for the detection a specific fault group using an algorithm. The output of ATPG is a set of test vectors. A test vector is an input vector for the CUT for detecting a subset of faults. If a test vector can be generated for a fault, it is called as a detected fault. Otherwise, the fault is undetected. Redundant faults cannot be detected by any test vector, mainly because of the redundancy in the logic.

An example for redundant faults is given in Fig. 1.8. The stuck-at-1 fault on net B is redundant and cannot be detected by any test vector under the single fault assumption. Detecting stuck-at-1 fault on net B requires forcing net B to logic value 0. B can be forced to 0 only if all inputs of the OR gate are forced to 0. However, forcing input C of the OR gate to 0 also forces input D of the AND gate to 0. Since logic value 0 is a controlling value for the AND gate, stuck-at-1 fault on net B is never observed (masked by D). As a result, stuck-at-1 fault on net B is redundant.

The success of ATPG is reported as fault coverage. Let $FC$ be the fault coverage, $\beta$ be the number of detected faults, and $\tau$ be the total number of faults in the CUT. Fault coverage is defined as the percentage of faults that can be detected using the generated test vectors:

\[ FC = \frac{\beta}{\tau} \]
Figure 1.8: A simple example of redundant fault. Stuck-at-1 fault is redundant on net B.

\[ FC = \frac{\beta}{\tau} \times 100 \]  

(1.1)

An alternative metric is fault efficiency. Let \( FE \) be the fault coverage and \( \alpha \) be the number of redundant faults. Fault efficiency is the percentage of faults that are detected or proven to be redundant:

\[ FE = \frac{\beta + \alpha}{\tau} \times 100 \]  

(1.2)

Commercial ATPG tools generally report the ATPG result as test coverage [15]. Let \( TC \) be the fault coverage. Test coverage is simply the percentage of detected faults over testable faults.

\[ TC = \frac{\beta}{\tau - \alpha} \times 100 \]  

(1.3)

The following terms are commonly used in ATPG literature:

**Controllability.** The ability to establish a specific signal value at each node by setting circuit’s inputs. Controllability is a metric to measure the difficult of setting a node to a specific logic value. The nodes that are easiest to control are the test insertion points. As we go deep into the circuit, the controllability of the circuit becomes more difficult. Controllability was first defined by Rutman.
in 1972 [16].

**Observability.** The ability to determine the signal value at any node in a circuit by controlling the circuit’s inputs and observing its output. Observability is a metric to measure the difficulty in propagating a signal value to primary outputs or test observation points. Observability was first defined by Goldstein in 1979 [17]. Later, Goldstein and Thigen introduced the SCOAP (Sandia Controllability and Observability Analysis Program) measure [18] which helped the terms of controllability and observability receive more attention.

**Sensitization.** The process of exciting a fault starting from the primary inputs. The term sensitization can also be used for paths in addition to a single fault. Path sensitization is the process of exciting the faults of an entire path starting from primary inputs to primary outputs.

**Propagation.** The process of propagating the fault effects to primary outputs.

**Justification.** The process of finding the input vector required to set a node a specific logic value.

A basic ATPG flow is shown in Fig. 1.9. The ATPG process begins by generating a list of all possible faults in the circuit under test. One of these faults is selected for test generation and ATPG is performed to find a test vector for this fault. If test vector generation is successful, in the next step, fault simulation is done to find the list of all of the faults that are detected by the generated test vector. The detected faults are removed from the overall fault list. If the test generation is not successful, the fault is dropped from the list but marked as redundant (if proven). Note that for
Figure 1.9: A basic ATPG flow.
most commercial ATPG tools, there is a search limit (usually a time or trial limit) while pattern generation. If the ATPG tool cannot find a test vector within the search limit, the fault is still dropped from the fault list, but marked as undetected. This process continues until all faults are dropped from the global fault list.

Many companies today rely on \( n \)-detect ATPG for more effective defect screening. The \( n \)-detect ATPG flow is similar to the basic ATPG flow. The difference is that each fault is aimed to be detected at least \( n \) times before dropping it from the overall fault list. The value added by \( n \)-detect ATPG is the efficiency in detecting unmodeled defects that would otherwise be missed by basic ATPG flows [19]. The drawbacks are the increased test pattern count and the longer pattern generation time.

### 1.1.4 Design for Testability

The purpose of design for testability (DFT) is to make the testing of the CUT easier and more efficient. Among many different DFT techniques, scan-based designs are the most common. In this section, we will briefly discuss the scan-based designs.

When the size of CUT (a.k.a. Design Under Test — DUT) is very large, e.g., today’s complex microprocessor designs, it becomes extremely difficult to control and observe fault locations. The deterministic ATPG time in such a case would be so long that it would not be feasible to find a good test set. This complex problem can be eased if we can insert test vectors at different internal locations in the CUT. Flip-flops are the best candidates for test insertion points due to their memory capability. Controllability and observability can be improved significantly by using scan registers. Scan registers are the flip-flops with special testing capability in addition to the ordinary flip-flop properties. Scan registers have the shift and
parallel load capability. When the circuit is in test mode, all scan flip-flops form one or more shift registers, referred to as scan chains [4]. In this way, a test vector can be shifted into the circuit using the scan registers. The response of the circuit is then shifted out in the same way. In a scan design, all scan register inputs are observation points and all scan register outputs are test insertion points.

Figure 1.10: A scan chain example.

Fig. 1.10 shows a simple scan chain example. The scan flip-flops are shown as white boxes. During the scan test, first the circuit is put into the test mode by using the scan enable (SE) signal. SE cuts the connection between the scan registers and the CUT, thus prevents the test vectors from stimulating the CUT during shift cycles. After SE is set to enable mode, the test vector is shifted into the scan registers. Then, SE is set disable mode and the test vector is applied to the CUT. The response of the CUT is captured by the scan registers. In the next step, SE is again set to enable mode and the response of the circuit is shifted out [14].

The scan operation slightly changes depending on the scope of the scan test. If the scan test targets stuck-at faults, it uses a single system clock pulse. However,
delay test requires double system clock pulse. The step-by-step details of a typical scan operations for stuck-at test can be described as follows:

1. (Fig. 1.11) The first step is to put the scan flip-flops into scan mode. This is done by using the Scan Enable signal. In this case, forcing Scan Enable to 1 enables the scan mode. Note that initially all the scan flip-flops at unknown state (X). For industrial circuits, there are architectural ways to initialize all flip-flops to known states if needed. However, for this particular case, we assume that all scan flops were initially at unknown state X.

Assume that the following test vector will be scanned-in: 100101011

2. (Fig. 1.12) The scan-in process starts. In Fig. 1.12, it can seen that the first 3 bits are scanned in. A single test bit is shifted-in at each clock cycle. Usually, the scan shift frequency is much slower than the functional operating frequency of the CUT. This frequency is currently about 100 MHz for most ASIC circuits, although there are companies using 400 MHz or above shift frequencies. Note that, the higher the test frequency, the shorter the test time. On the other hand, higher shit frequency requires better (more expensive) testers.

3. (Fig. 1.13) At this stage, the complete test vector 100101011 is shifted-in. Scan mode can be disabled by forcing Scan Enable to 0. Note that the shifted-in test vector is currently applied to the combinational logic pieces that are driven by scan flip-flops. It means that 2\textsuperscript{nd}, 3\textsuperscript{rd}, and 4\textsuperscript{th} combinational logic blocks are already forced test inputs.

4. (Fig. 1.14) The next step is to force primary input (PI) values and measure
the primary output (PO) values: force\_PI and measure\_PO.

Note that from the previous step, the shifted-in test vector was already applied to the combinational logic pieces that were driven by scan flip-flops. It means that 2\textsuperscript{nd}, 3\textsuperscript{rd}, and 4\textsuperscript{th} combinational logic blocks were already forced test inputs. Now, these combinational logic blocks have generated their outputs.

Since test inputs are forced to PI, the 1st combinational block also has its outputs ready. Furthermore, the outputs of the 4th combinational block can now be observed from POs. Output values of combinational block 4 can be obtained by measuring POs.

For the rest of the combinational blocks (1, 2, and 3), we need to push the output values into scan flip-flops and then shift these values out.

\textbf{Note:} In reality, for industrial circuits, force\_PI and measure\_PO is not done. This is because primary inputs and outputs are connected to very slow pads, and these pads are not tested by structural test. You may realize that in this case the 1\textsuperscript{st} and 4\textsuperscript{th} combinational blocks cannot be tested: 1\textsuperscript{st} block cannot be tested because test inputs cannot be applied to it (force\_PI). 4\textsuperscript{th} block cannot be tested because its outputs cannot be observed (measure\_PO). This is usually not a problem because the circuits are surrounded by wrapper scan flip-flops. This means that there is actually no significant logic before the first level of scan flip-flops or after the last level of scan flip-flops. As a result, almost the complete CUT is covered by scan flip-flops.

5. (Fig. 1.15) In order to push the output values of combinational blocks 1, 2, and 3 into scan flip-flops, the system clock should be toggled. Once this is done,
all D flip-flops (scan flip-flops) will capture the values at their D input. In Fig. 1.15, the capture event is shown.

6. (Fig. 1.16) CUT is ready to shift-out the captured combinational logic responses. However, while doing that, the next test vector is also shifted-in. The next test vector is shown in Fig. 1.16 as $111100111$. Note that the Scan Enable signal is set back to 1 to enable shifting.

7. (Fig. 1.17) A snapshot of the shift operation is shown in Fig. 1.17. As seen, 4-bits of the previous test response have been shifted-out, and at the same time 4-bits of the new test vector input have been shifted-in. The new test vector bits are shown in bold-red in Fig. 1.17.

8. (Fig. 1.18) At this stage, the test response for the previous test vector is completely scanned-out (shifted-out), and the new test vector input is also scanned-in (shifted-in). The process continues in this way until all the test vectors are applied.

Scan operation for delay test is very similar to stuck-at test. The main difference is that delay test needs two inputs instead of one. The first input is always the scanned-in vector. The second input can be generated in two different ways. Each way has its own name: (1) Launch-on-Capture (LOC) or broadside delay test, (2) Launch-on-shift (LOS) or skewed-load delay test. The examples below will show how each of these methods works.

The step-by-step details of a typical scan operations for LOC delay test can be described as follows:
1. (Fig. 1.19) The initial steps are the same as stuck-at test (Fig. 1.11-1.14) as described above. Fig. 1.19 shows the state of CUT after initial steps. The test vector is scanned-in, PIs are forced, and they created some output responses for combinational blocks. The first vector for the delay test is already applied. Now, it is required to create a second test vector to create signal transitions. The second vector will be the output responses of the combinational blocks. Each block will generate the second test vector for the next stage. Since there is no stage before the 1st combinational block, force PI needs to be applied one more time.

2. (Fig. 1.20) In order to push the output responses of combinational blocks into scan flip-flops, the system clock is toggled. Once this is done and second PI force is applied, the second test vector for the delay test is generated. As a result, each combinational circuit input will see an input state transition. The transition on scan flip-flop outputs — which are inputs to the next stage combinational block — will be as follows (starting from the closest-to-scan-in flop): 100101110 --> 010010111

The second input vector will generate output responses similar to the first one. These output responses needs to be captured, similar to the first one, by toggling the system clock. However, now there is a difference: The system clock has to be toggled at the real operating frequency. This means that the period between the first clock toggle and second clock toggle should be equal to functional clock period. In this way, the delay-test responses are captured at the functional operating frequency. As a result, correct functionality of the
circuit is tested at-speed.

3. (Fig. 1.21) A timing-diagram of the LOC process is shown in Fig. 1.21. As seen, the test vector is shifted using a slow clock frequency. Then, the scan enable is set to 0 and scan mode is disabled. In the next step, the clock is toggled first time to launch a transition in combinational blocks. After that, it is toggled again (at the functional frequency) to capture the final responses of the combinational blocks. The launch and capture events happen at functional frequency. Finally, the captured responses are shifted-out using the slow clock frequency.

The step-by-step details of a typical scan operations for LOS delay test can be described as follows:

1. (Fig. 1.22) The initial steps are the same as stuck-at test (Fig. 1.11-1.14) as described above. Fig. 1.22 shows the state of CUT after initial steps. The test vector is scanned-in, PIs are forced, and they created some output responses for combinational blocks. For LOS, these initial output responses are not captured. The first vector for the LOS delay test is already applied. Since there is no stage before the 1st combinational block, a second force PI is applied. Note that scan enable signal is still at active value 1. This is because shifting will continue. One more bit needs to be shifted-in to create the second test vector for the LOS delay test.

2. (Fig. 1.23) Note that as mentioned in the previous page, the first vector of the delay test is (starting from the closest-to-scan-in flop) 100101110. The second
test vector is generated by shifting one more time, and inserting one more bit from Scan In. Hence, the second test vector is 010010101. Just after shifting-in the last bit (and launching a transition by applying the second vector), scan enable has be forced to 0, and the system clock should be toggled at the functional frequency. The last toggle of the system clock captures the delay-test responses. Finally, the final responses are scanned-out as usual.

3. (Fig. 1.24) A timing-diagram of the LOC process is shown in Fig. 1.24. As seen, the test vector is shifted-in using a slow clock frequency until the last bit. The last shifted bit creates the launch event. Then, before toggling the system clock to capture responses, scan enable is set to 0 and disable scan mode. This has to happen very fast since launch and capture events happen at high frequency. In the next step, the clock is toggled again to capture the final responses of the combinational blocks. Finally, the captured responses are shifted-out using the slow clock frequency.

From the above discussion, we note that is necessary to have a very fast scan enable signal in order to use LOS. Scan enable should be able to switch from 1 to 0 within a very short time. This is usually difficult to accomplish because scan enable is not designed to operate at high frequencies. There are other advantages of LOC over LOS:

- LOC has a higher defect-detection probability. This is mainly because the transition in LOC patterns are generated by the circuit itself, LOC patterns will catch defects not only in the fanout cone of scan flip-flops, but also the fan-in cone of these flip-flops. A defect in a previous stage
Figure 1.11: Scan test is initialized by forcing scan enable signal.

may prevent a correct transition in the next stage.

- LOC patterns are less likely to create functionally impossible states. This is again because the transition in LOC patterns are generated by the circuit itself. On the other hand, LOS patterns are generated by a simple shift in the scan chain. This means that any transition becomes possible, even though it is functionally impossible.

Due to the reasons mentioned above, many industrial designs use LOC instead of LOS. However, there are some designs that use LOS, and there are proposed workarounds to fast scan enable signal requirement.

The test vectors are applied to the CUT by an automatic test equipment (ATE). The responses are also stored and analyzed by the ATE. ATE is provided with the good circuit response, i.e., the expected response, to determine whether the CUT is fault or not. ATEs are complicated and very expensive tools with a limited memory to apply test vectors and keep responses [20]. They are the main contributors to the
Figure 1.12: Shift-in operation started.

Figure 1.13: The test vector is completely shifted-in.
Figure 1.14: PIs are forced and combinational blocks generated outputs.

Figure 1.15: System clock is toggled.
Figure 1.16: Combinational block outputs are captured in scan flip-flops. The next test vector is ready to be shifted-in.

Figure 1.17: While captured responses of the previous vector is shifted-out, the new test vector is shifted-in.
Figure 1.18: The responses of the previous vector is completely shifted-out, the new test vector is completely shifted-in.

Figure 1.19: Initial steps for LOC delay test generates the same responses as stuck-at test.
Figure 1.20: Second test inputs are generated by a second PI force and a clock toggle.

Figure 1.21: Timing diagram for LOC delay test.
Figure 1.22: LOS delay test starts like the LOC delay test.

Figure 1.23: Second test inputs are generated by shifting scan chain values.
overall test cost. The test methods that can detect a defective part in the shortest time using the smallest possible set of test vectors is desirable mainly because of these reasons.

### 1.1.5 Test Data Compression

Over the last decade, as the designs grew larger (in the range of several millions of logic gates or more), the memory limitation of the ATE emerged as a bottleneck. Moreover, the test application time has increased considerably [21]. Test data compression refers to the method of compressing the test vectors before test application and decompressing them in the CUT using a decompressor. The response of the CUT can also be compressed before it is sent to the ATE. Typically more than 10x, and as often as 100x, compression can be obtained using today’s compression tools [22].

Test data compression schemes can be divided into three groups [23]:

- **Code-based schemes:** Test data is compressed by partitioning the data into
segments, and replacing segments by shorter code bits [24–27].

- **Linear-compressor-based schemes**: Test compression is provided by XOR trees and/or linear-feedback shift-registers (LFSRs) [22]. By leveraging the fact that most of the test vectors actually includes a high percentage of *don’t-care* bits, the problem of test data compression is transformed into a linear algebra problem. A small number of seeds are found by solving a system of linear equations, which can be used to generate test vectors driving larger number of scan chains. An example of this scheme is presented in Fig. 1.25. In this example, the LFSR and the phase shifter forms the on-chip test decompressors. The ATE transfers the seeds into the decompressor block, and the decompressor generates patterns for a large number of internal scan chains. The outputs of internal scan chains are again compressed by an XOR tree and fed back to ATE for checking.

- **Broadcast-scan-based schemes**: A small number of external test pins are used to drive a larger number of internal scan chains. As a result, each test pin broadcasts the test data to multiple scan chains. Connections between the internal scan chain input and external test pins are dynamically changed by using multiplexors to achieve maximum test compression [28].

**How does compression work? An example**: A typical LFSR-reseeding-based compression scheme can be described in an example as follows:

- Assume that DUT has 100 internal scan chains and all of them are of length 50 (there are 50 scan flip-flops on each of them).
Figure 1.25: A high level view of the LFSR-reseeding-based test data compression scheme.
• Also, assume that there are 5 input test channel pins available. It means that using these 5 input pins, the decompressor will generate patterns for 100 scan chains.

• Without getting into further detail, it can be seen that test time will be saved: If there are 5 input channels, in means that there are only 5 pins available for scan-test input. If test compression is not used, DUT can only have 5 scan chains. There are a total of \(100 \times 50 = 5000\) scan flip-flops. If DUT has 5 chains, each chain has to be of length 1000. However, when test compression is used, the scan chain length is 50. It means that each pattern will take much less time to shift-in and shift-out. As a result, considerable amount of test time can be saved.

• Now, assume that the DUT employs an LFSR-reseeding-based test-compression scheme. At each shift-in cycle, a single “seed” is pushed into the on-chip test decompressor. This seed is 5 bits long. For each of these 5-bit seeds, decompressor will generate 100-bits to feed internal scan chains.

• The ATPG tool generates test vectors in terms of test cubes. Each test cube targets a small portion of the DUT, hence a large percentage of the test cube includes don’t care (\(X\)) bits. Assume that the ATPG tool generated the following test cubes:

\[\begin{align*}
\text{Chain}_0 : & \quad XXXXX1XXX0XXXXXX0101XXX...
\text{Chain}_1 : & \quad X1XXX0XXXXXXXXXX01X0XXX...
\text{Chain}_2 : & \quad XXXXXXXXXXX0XXXXXX0XXXXXX...
\end{align*}\]
\[ Chain_3 : 0XXXXXXX1XXXXXX0XXXXXX \ldots \]
\[ Chain_4 : X1XX1XX0XXXXXX0XXXXXX \ldots \]
\[ Chain_5 : XXXXX011XXXXXX0XXXXXX \ldots \]
\[ Chain_6 : 1XX10XXXXXXX0XXXXXX \ldots \]

- In the first shift-in cycle, internal scan chains need to be fed by (first bit of the each test cube given above):

\[ XXX0XX1 \ldots \]

This test vector has a length of 100-bits (for 100 internal scan chains). It includes mostly \( X \)'s, and only a few specified bit values. A compression algorithm solves systems of linear equations to find the seed value for this specific vector. Note that the solver only needs to satisfy the specified bits, and \( X \)'s can be mapped to any value. The computed 5-bit seed value is shifted-in to the input channels.

If no test compression is used, the ATPG-tool-generated test cubes still contain many \( X \)'s. However, in that case, the ATPG tool either tries to fill these \( X \)'s with some random values and/or tries to merge multiple cubes into a single one. As a result, the final test vectors are fully-specified. This scheme is called static compaction, and it is another way of compressing test data. However, higher test-time and test-volume reduction can be achieved by hardware test-data compression schemes.
1.2 Motivation for Thesis Research

With continuously scaling in process technology, we are now in the age of nanometer ICs (Fig. 1.26). However, advances in technology have also led to high defect rates. Today’s very deep sub-micron (VDSM) designs, which lead to high chip densities and every higher clock frequencies, are especially prone to physical defects and process variations, which lead to excessive circuit delay and unpredictable delay variations.

![Figure 1.26: Transistor gate length over the years [2].](image)

We can summarize the common problems of nanometer-technology designs as follows [1]:

- More bridging faults and resistive shorts are observed.
- More resistive open defects are observed. Package fault analysis results show that opens and resistive opens account for 58% of all defects.
• As technology scales down, wires get closer and longer; crosstalk becomes more dominant.

• Circuit performance becomes highly sensitive to process variations.

All these effects are noticeable in today’s technologies and they are likely to become more prominent in the next-generation process technologies [1]. A common feature of all these effects is that they all contribute to small-delay defects (SDDs). SDDs are the small delay variations, more frequent than gross-delay defects (Fig. 1.27 [3]), which are induced by the above listed effects in the circuit components [29,30].

1.2.1 Difficulty of Detecting SDDs

The detection of SDDs is difficult because of the small size of the introduced delay. Although the delay introduced by each SDD is small, the overall impact can be significant if the target path is critical, has low slack, or includes many SDDs. The overall delay of the path may become larger than the clock period, causing circuit failure or temporarily incorrect results. As a result, the detection of SDDs typically requires fault excitation through least-slack paths. The longest paths in the circuit, except false paths and multi-cycle paths, are referred to as the least-slack paths.

Widely-used ATPG techniques are not effective at exciting small delay defects. Stuck-at test is not sufficient for high quality test, and it will most likely miss any delay defect which is not a gross-delay defect. The traditional transition-test ATPG, on the other hand, is inclined to select short activation and propagation paths due to the driving SCOAP measure [18]. However, SDDs are observable only on short-slack
paths. Furthermore, major contributors to SDDs are wire delay variations and wire defects, which are ignored by traditional delay test methods. New ATPG methods are therefore necessary to exercise short-slack paths.

### 1.2.2 The Need for Layout-aware ATPG

Interconnects are expected to be a major performance limiter when process technologies shrink to 45nm and below [31,32]. This is mainly because the technology scale-down in interconnects, unlike transistors, does not always lead to higher performance. ITRS predicts that the wire-delay relative to the gate-delay will increase considerably for smaller technology nodes. Inductive effects due to increasing clock frequencies are expected to push interconnect delays to even higher values, which will force industry to adopt new interconnect technologies [32].

Process variation on wires have a greater effect on circuit delay than process variation for transistors. Studies predict that process variations on interconnects will have a considerable impact on critical- and long-path delays [32–34]. The reason behind these findings is that interconnects are not only the dominant contributor
to overall path delays, they are also susceptible to large process variations due to lithography effects, etching, gradients, and various random effects [35]. For instance, as much as 40% wire-width variation is estimated for wires of 40nm width [33].

Technology shrinking is expected to have different effects on interconnects at various metal layers. For technologies below 45nm, Metal-1 and intermediate-level metal wires (local wires) are expected to have the same line-widths and thicknesses, thus they will show similar process variation effects. This is mainly because these wires usually shrink when traditional scaling is applied to transistors. The effect of process variations on local metal layers are expected to have smaller effect on overall circuit delay because of their relatively shorter lengths compared to global wires [32, 33]. Global wires, on the other hand, are expected to have increasing nominal delays and larger impact on overall circuit delay due to process variations. A recent study in [34] shows considerable increase in the variation of $RC$ delays as the wire length increases. In addition to metal layers, vias are also affected by process variations. The resistance variation of vias is larger compared to that for metal layers because they tend to have the smallest possible dimensions [35].

Process variations are not the only parameters affecting the interconnect delay, and as a result path delays. Crosstalk effects on wires also have significant impact on a circuit’s timing uncertainty [36]. The coupling-line length that can change the voltage levels by 25% of the supply voltage on a victim wire due to crosstalk is expected to drop 40% by 2013 [31, 32]. This implies that for the same wire length, there will be much stronger crosstalk effects in the future. Furthermore, at multi-gigahertz clock frequencies, because of the introduction of inductive coupling, crosstalk is expected to have 60% greater impact than predicted by $RC$ models [31].
In view of the above trends and projections, delay-test generation methods that are oblivious of interconnect and layout-related effects are not adequate for next-generation technologies. As a result, a number of new delay-fault pattern generation techniques have been developed [29,37–39]. However, most of the proposed methods are aimed at finding the longest paths in a circuit, rather than being driven by layout and variation data.

1.2.3 Limitations of Existing Commercial Timing-aware ATPG Tools

Due to the growing interest in SDDs, the first commercial timing-aware automatic test-pattern generation (ATPG) tools were introduced recently, e.g., new versions of Mentor Graphics FastScan, Cadence Encounter Test, and Synopsys TetraMax [40–42]. These tools attempt to make ATPG patterns more effective for SDDs by exercising longer paths or applying patterns at higher than rated clock frequencies. However, only a limited amount of timing information is supplied to these tools, either via standard delay format (SDF) files (for FastScan and Encounter Test) or through a static timing analysis (STA) tool (for TetraMax). As a result, none of these tools can be easily extended to take into account process variations, crosstalk, power-supply noise, or similar SDD-inducing effects on path delays. These tools simply rely on the assumption that the longest paths (determined using STA or SDF data) in a design are more prone to failure due to SDDs. Moreover, the test generation time increases considerably when these tools are run in timing-aware mode. Fig. 1.28 shows a comparison of the run times of two ATPG tools from the same EDA company: (i) timing-unaware ATPG, i.e., a traditional transition-delay-
fault pattern generator and (ii) timing-aware ATPG that takes timing information into account. The results are shown for some of the IWLS’2005 benchmarks [43] and the absolute run times are shown in Table 2.5. It can be seen from Fig. 1.28 that, when the benchmark is large, the timing-aware ATPG takes significantly higher CPU time, e.g., as much as 209x more for the “netcard” benchmark. The CPU times in Figure 1.28 reflect distributed ATPG results using eight processors, and the numbers are normalized such that the run-time of timing-unaware ATPG is taken to be one unit. Similar results are obtained on industrial circuits. Fig. 1.29 shows the relative CPU time and pattern count of timing-aware ATPG compared to a traditional transition-delay-fault pattern generator. As seen, as much as 22x CPU time increase and 15x pattern count increase is observed. These results show that currently available commercial timing-aware ATPG is costly in both CPU time and more importantly in the pattern count inflation. As a result, it is crucial to find a small subset of patterns without losing the pattern quality.

1.2.4 The Need for Reducing Test Data Volume

The complexity of today’s ICs and shrinking process technologies are also leading to prohibitively high test-data volumes. For example, the test data volume for transition-delay faults is 2-5 times higher than that for stuck-at faults [44], and it has been demonstrated recently that test patterns for such sequence- and timing-dependent faults are more important for newer technologies [45]. The 2007 International Technology Roadmap for Semiconductors (ITRS) predicted that the test data volume for integrated circuits will be as much as 38 times larger and the test application time will be about 17 times larger in 2015 than it was in 2007 [1]. Therefore,
efficient pattern-selection methods are required to reduce the total pattern count while effectively targeting SDDs.

Test-pattern reordering methods, which rank test patterns and place the most effective test patterns at the top of the reordered test sequence, promise reductions in both testing time and test data volume [46–48]. If highly effective test patterns are applied first in a reordered test set, defective chips will fail earlier, reducing test application time in an abort-at-first-fail environment. The reordered test set can be simply truncated to fit test-data-volume and meet test-time budgets.
1.2.5 Summary of Research Needs

The above discussion highlights the pressing need for improved, layout-aware pattern generation methods that can handle delay variation on interconnects. These methods must be able to identify high-quality delay-test patterns with low CPU times from a large repository of candidate test patterns. Some early work has been reported in this direction [49–51], but none of these techniques handle variation on interconnects. This thesis work is an attempt to fill this void by making the output deviations metric from [51] cognizant of interconnect and layout.

This thesis uses the output deviation measure [48, 52] as a surrogate coverage-metric for SDDs and a test-pattern grading method to select the best patterns for SDD detection from a large repository test set. A flexible, but general, probabilistic fault model is used to generate a probability map for the circuit, which is subsequently

Figure 1.29: CPU time and pattern count of timing-aware ATPG relative to traditional TDF ATPG for sub-blocks of an industrial microprocessor circuit.
used for pattern reordering. The proposed method can be used with traditional, “no-timing” ATPG tools to generate a high-quality delay-fault pattern set. We start with an initial set of patterns (without loss of generality, we consider $n$-detection transition-fault test patterns) and apply our pattern-grading method to calculate output deviations for each pattern. We then sort the patterns according to their ability to detect SDDs effectively. We also show how the pattern-ordering method can be used to truncate large timing-aware test sets.

### 1.3 Overview of Related Work

SDDs were first alluded to in [53]. In recent years, high-quality delay-fault pattern generation for SDDs has received increasing attention. Most of the work is aimed at finding the longest paths in a circuit. Gupta et al. [39] have proposed the “As Late As Possible Transition (ALAPTF)” fault model, which attempts to launch one or more transitions at the fault site as late as possible, i.e., through the least-slack path using robust tests. This method suffers from the need for a complex, time-consuming search procedure part and robust test-generation constraints. Qui et al. [38] attempt to find the $k$-longest paths (referred to as KLPG) through the inputs and output of each gate for slow-to-rise and slow-to-fall faults. Similar to [39], a considerable amount of pre-processing is needed to search for long paths. Furthermore, a long path through a gate may be a short path in the circuit, thus not all the paths determined by the method are least-slack paths. Ahmed et al. [29] use static timing analysis tools to find long (LP), intermediate (IP), and short paths (SP) to each observation point. Using a timing-unaware ATPG tool, 15-detect transition test patterns are generated. During
pattern generation, constraints are applied on IP and SP observation points to mask them. In this way, the ATPG tool is forced to generate patterns for LPs. In the post-processing phase, a pattern-selection algorithm is used to pick patterns that activate the largest number of end-points. Similar to previous methods, a time-consuming search procedure is needed for determining long paths and for path classification. A functional delay fault test generation method is proposed in [54]. This method generates sequences of instructions for testing delay faults. However, it requires a fault-free unit that can run the instructions for the test program. Similar to earlier methods, this scheme also involves a lengthy preprocessing step. In [55], delay defects within slack intervals are detected by using a clock frequency higher than the rated clock frequency. This method uses a good neighboring die to test the surrounding dies. The responses of the good die and the other dies are compared with each other to find delay defects. A new fault model, called the “Transition Path Delay Fault Model” is described in [56]. This fault model relaxes the robustness constraint required by the path-delay fault model, and aims to excite paths that are missed by the path-delay fault model. [57] presents a method to accurately determine the fault coverage of path-delay tests by analyzing path reconvergences. This method is applicable to the bounded gate-delay model.

As a result of increasing industry concern regarding SDDs, companies such as Mentor Graphics, Cadence, and Synopsys have recently released timing-aware ATPG tools [30,40–42]. Lin et al. [30] use SDF files to guide ATPG to generate transition test through long paths. In a pre-processing step, the proposed method evaluates the delay for “activation” and “propagation” paths for each gate to find longest paths. Test generation is guided by the results of this pre-processing step. Although
approximation methods are used to decrease the overhead associated with delay and path-length calculations, this method still takes considerably more time compared to timing-unaware ATPG tool. Kapur et al. [42] uses STA tool generated pin slack information to guide timing-aware ATPG. Although the preprocessing step is skipped by pushing the slack data calculation to the STA tool, pattern generation takes considerably more time than timing-unaware ATPG.

Statistical static timing analysis (SSTA) can generate variability-aware delay data. However, as clearly demonstrated in recent papers [58, 59], SSTA does not find sensitized paths based on input vectors using statistical data.

The “number of activated long paths” is a useful metric for evaluating the quality of delay-fault pattern quality, but a more computationally-tractable method is clearly needed. An alternative evaluation method, referred to as the “statistical delay quality model” (SDQM) has been proposed by Sato et al. [3]. This pattern-grading metric is based on a delay-defect distribution function, which requires delay-defect statistics for fabricated ICs. The method assigns a “statistical delay quality level (SDQL)” to each test set to evaluate its quality. A drawback of this metric is the need for delay-defect distributions for real chips. This data is not available before volume production and it is difficult and very expensive to obtain it during production test. Another shortcoming of SDQM and similar metrics is that they require knowledge of the longest sensitizable paths, which is not accurately known before production test.
1.4 Thesis Outline

The remainder of the thesis is organized as follows. In Chapter 2, we introduce the output-deviations metric for SDDs and present the framework for targeting gate-delay variations. In Chapter 3, we show how variations in wire delays can be incorporated in the output deviations framework. Chapter 4 presents the enhanced method for the consideration of crosstalk effect on wire delay variations. In Chapter 5, we present how to use the output-deviation metric in an LFSR-reseeding-based test compression technique to detect SDDs. Chapter 6 shows the applications of methods presented in previous chapters to some representative industrial circuits. Chapter 7 includes concluding remarks and future research directives.
Chapter 2

Test-Pattern Grading and Pattern Selection

In this chapter, we present the output deviation measure [48,51,52] as a surrogate coverage-metric for SDDs and a test-pattern grading method to select the best patterns for SDD detection from a large repository test set [51]. A flexible, but general, probabilistic fault model is used to target defects. The proposed method can be used with traditional, timing-unaware ATPG tools to generate a high-quality and compact delay-fault pattern set. It can also be used to select the most-effective patterns from large timing-aware test sets. Experimental results show that the proposed method can effectively select the highest-quality patterns from large test sets that cannot be used in entirety for production test environments with tight pattern-count limits. It also considers process-variability-induced delay variations, unlike most previous methods. The proposed approach requires significantly less CPU time than a commercial timing-aware ATPG tool under pattern-count limits. For various metrics, namely coverage of long paths, detection of injected defects, and coverage ramp-up, it is shown to outperform a commercial timing-aware ATPG tool. We also compare the proposed method with the approach proposed by Lee et al. [60], where path-length calculations are approximated for better run-time, and we highlight better long-path coverage, lower run times, and faster coverage ramp-up for injected faults.

The key contributions of this chapter are as follows:
• We show how the previously proposed output-deviations metric [48,52] can be reformulated to account for SDDs.

• We present the underlying theory and algorithms used to calculate output deviations.

• We develop the framework to consider process variation effects in testing for SDDs.

The remainder of this chapter is organized as follows. In Section 2.1, we describe the probabilistic fault model and the output deviations metric. Section 2.1.4 presents the proposed pattern-selection procedures. In Section 2.2, we evaluate the proposed method for benchmark circuits and n-detection TDF test sets. We also conduct simulated defect-injection experiments to evaluate the effectiveness of the selected patterns for detecting small delays caused by resistive shorts and opens. Section 2.3 summarizes this chapter.

2.1 Probabilistic Delay-Fault Model and Output Deviations for SDDs

In this section, we first introduce the concept of gate-delay defect probabilities (DDPs) (Section 2.1.1) and signal-transition probabilities (Section 2.1.2). These probabilities extend the notion of confidence levels, defined in [48] for a single pattern, to pattern-pairs. Next, we show how to use these probability values to propagate the effects of a test pattern to the test observation points (scan flip-flops/primary outputs) (Section 2.1.2). We describe the algorithm used for signal-probability prop-
agation (Section 2.1.3). Finally, we describe how test patterns can be ranked and selected from a large repository (Section 2.1.4).

2.1.1 Gate-Delay Defect Probabilities

DDPs are assigned to each gate in a design. DDPs for a gate are provided in the form of a matrix called the *Delay Defect Probability Matrix* (DDPM). The DDPM for a 2-input OR gate is shown in Table 2.1. The rows in the matrix correspond to each input port of the gate and the columns correspond to the initial input state during a transition.

Assume that the inputs are shown in the order of \( \text{IN0, IN1} \). If there is an input transition from ‘10’ to ‘00’, the corresponding DDPM column is ‘10’. Since the transition is caused by \( \text{IN0} \), the corresponding DDPM row is \( \text{IN0} \). As a result, the DDP value corresponding to this event is 0.5. 0.5 shows the probability that corresponding output transition is delayed beyond a threshold.

For initial state ‘11’, both inputs should switch simultaneously to have an output transition. Corresponding DDPM entries are merged due to this requirement. The entries in Table 2.1 have been chosen arbitrarily for the sake of illustration. The real DDPM entries are much smaller than the ones shown in this example. Read DDPM entries can be found in Appendix A.

Table 2.1: Example DDPM for a 2-input OR gate

<table>
<thead>
<tr>
<th>OR</th>
<th>prob</th>
<th>Initial Input State</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN0</td>
<td>0.5</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>IN1</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>11</td>
</tr>
<tr>
<td>IN0</td>
<td>0.2</td>
<td>00</td>
</tr>
<tr>
<td>IN1</td>
<td>0.2</td>
<td>01</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>11</td>
</tr>
</tbody>
</table>
For an $N$-input gate, the DDPM consists of $N \cdot 2^N$ entries, each holding one probability value. If the gate has more than one output, each output of the gate has a different DDPM. Note that the DDP is 0 if the corresponding event cannot provide an output transition. Consider DDPM(2,3) in Table 2.1. When the initial input state is ‘10’, no change in $IN_1$ can cause an output transition, because the OR gate output is already at high state, and even if $IN_1$ switches to high (1), this will not cause an output transition.

We next discuss how a DDPM is generated. Each entry in DDPM indicates the probability that the delay of a gate is more than a predetermined value, i.e., the critical delay value ($T_{CRT}$). $T_{CRT}$ is an instance dependent value. Fig. 2.1 shows an example Gaussian delay distribution curve and $T_{CRT}$-DDP relation.

If we assume a Gaussian delay distribution for all gates (with mean $\mu$) and set the critical delay value to $\mu + \tau$, each DDP entry can be calculated by computing the area enclosed by the Gaussian pdf curve and $T_{CRT}$ line. Note that the delay for each input-to-output transition may have a different mean ($\mu$) and standard deviation ($\sigma$). The selection of $\tau$ is described in Section 2.2.1. Note that the delay distribution
Figure 2.2: An example non-Gaussian delay distribution curve to illustrate $T_{CRT}$ and DDP. The shaded area corresponds to the defect region (DDP).

can be non-Gaussian as shown in Fig. 2.2. The definition of DDP only requires an area calculation independent of the shape of the delay distribution curve.

The delay distribution can be obtained in different ways: (i) using the delay information provided by an SSTA generated Standard Delay Format (SDF) file; (ii) using slow, nominal, and fast process corner transistor models; (iii) simulating process variations. In the third method, which is employed in this chapter, transistor parameters affecting the process variation and the limits of the process variation (3σ) are first determined. Monte Carlo simulations are next run for each library gate under different capacitive loading and input slew rate conditions. Once the distributions are found for the library gates, depending on the layout, the delay distributions for each individual gate can be updated. Once the distributions are obtained, $T_{CRT}$ can be appropriately set to compute the DDPM entries. The effects of crosstalk can be simulated separately and the delay distributions of individual gates/wires can be updated accordingly.

The generation of the DDPMs is not the main focus of this thesis. We consider DDPMs to be analogous to timing libraries. Our goal is not to develop the most
effective techniques for constructing DDPMs; rather, we are using such statistical
data to compute deviations and use them for patterns grading and pattern selection.
In a standard industrial flow, statistical timing data can be developed by specialized
timing groups, so the generation of DDPMs is a pre-processing step and an input to
the ATPG-focused test-automation flow.

We have also seen that small changes in the DDPM entries have negligible impact
on the pattern-selection results. We attribute this finding to the fact that any DDPM
changes affect multiple paths in the circuits, hence their impact is amortized over
the circuit and the test set. The absolute values of the output deviations are less
important than the relative values for different test patterns. Detailed results are
presented in Section 2.2.

2.1.2 Propagation of Signal-Transition Probabilities

Since pattern pairs are required to detect TDFs, there can be a transition on each
net of the circuit for every pattern-pair. If we assume that there are only two possible
logic values for a net, i.e., LOW (L) and HIGH (H), the possible signal transitions are
$L \rightarrow L$, $L \rightarrow H$, $H \rightarrow L$, and $H \rightarrow H$. Each of these transitions has a corresponding
probability, denoted by $P_{L \rightarrow L}$, $P_{L \rightarrow H}$, $P_{H \rightarrow L}$, and $P_{H \rightarrow H}$, respectively, in a vector
form ($< ... >$): $< P_{L \rightarrow L}, P_{L \rightarrow H}, P_{H \rightarrow L}, P_{H \rightarrow H} >$. We refer to this vector as the
signal-transition probability (STP) vector. Note that $L \rightarrow L$ or $H \rightarrow H$ implies that
the net keeps its value, i.e., no transition occurs.

The nets that are directly connected to the test-application points are called
initialization nets (INs). These nets have one of the signal-transition probabilities,
corresponding to the applied transition test pattern, equal to 1. All the other signal-
transition probabilities for INs are set to 0. When signals are propagated through several levels of gates, the signal-transition probabilities can be computed using the DDPM of the gates. Note that interconnects can also have DDPMs to account for crosstalk. In this chapter, due to the lack of layout information, we only focus on variations impact on gate delay. The overall deviation-based framework is, however, general and it can easily accommodate interconnect delay variations if layout information is available, as has been reported in [61].

**Definition 2.1.** Let $P_E$ be the probability that a net has the expected signal-transition for a given pattern-pair. The deviation on that net is defined by $\Delta = 1 - P_E$. The following rules are applied during the propagation of signal-transition probabilities:

1. If there is no output-signal transition (output keeps its logic value), then the deviation on the output net is 0.

2. If there are multiple inputs that can cause the expected signal-transition at the output of a gate, only the input-to-output path that causes the highest deviation at the output net is considered. The other inputs are treated as they have no effect on the deviation calculation (i.e., they are held at the non-controlling value).

3. When multiple inputs are required to change at the same time in order to provide the expected output transition, all required input-to-output paths of the gate are considered. Only the unnecessary (redundant) paths are discarded.

A key premise of this chapter is that output deviations can be used to compare path lengths. As in the case of path delays, the net deviations also increase as
the signal propagates through a sensitized path, a property that follows from the rules used to calculate signal-transition probability for a gate output. This claim is formally proven next.

**Lemma 2.1.** For any net, let the STP vector be given by \(<P_{L\rightarrow L}, P_{L\rightarrow H}, P_{H\rightarrow L}, P_{H\rightarrow H}>\). Among these four probabilities, i.e., \(<P_{L\rightarrow L}, P_{L\rightarrow H}, P_{H\rightarrow L}, P_{H\rightarrow H}>\), at least one is non-zero and at most two can be non-zero.

**Proof:** If there is no signal-value change (the event \(L \rightarrow L\) or \(H \rightarrow H\)), the expected signal-transition probability is 1 and all other probabilities are 0. If there is a signal-value change, only the expected signal-transition events and the delay-fault case have non-zero probabilities associated with them. The delay-fault case for an expected signal value change of \(L \rightarrow H\) is \(L \rightarrow L\) (the signal value does not change because of a delay-fault). Similarly, the delay-fault case for an expected signal value change of \(H \rightarrow L\) is \(H \rightarrow H\).

**Theorem 2.1.** The deviation on a net always increases or stays constant on a sensitized path if the signal-probability propagation rules are applied.

**Proof:** Consider a gate with \(K\) inputs and one output. The signal-transition on the output net depends on one of the following cases. From Lemma 1, we note that only two cases need to be considered.

---

**Figure 2.3:** Examples to illustrate the propagation rules.
(i) Only one of the input-port signal-transitions is enough to create the output signal-transition.

(ii) Multiple input-port signal transitions are required to create the output signal-transition.

Let $P_{OUT,j}$ be the probability that the gate output makes the expected signal transition for a given pair of patterns on input $j$, where $1 \leq j \leq K$. Let $\Delta_{OUT,j} = 1 - P_{OUT,j}$ be the deviation for the net corresponding to the gate output.

**Case (i):** Consider a signal transition on input $j$. Let $Q_j$ be the probability of occurrence of this transition. Let $d_j$ be the entry in the gate’s DDPM that corresponds to the given signal transition on $j$. The probability that the output makes a signal transition is given by:

$$P_{OUT,j} = Q_j(1 - d_j).$$  \hspace{1cm} (2.1)

We assume here that an error at a gate input is independent from the error introduced by the gate. Note that $P_{OUT,j} \leq Q_j$ since $0 \leq d_j \leq 1$. Therefore, the probability of getting the expected signal transition decreases and the deviation $\Delta_{OUT,j} = 1 - P_{OUT,j}$ increases (or does not change) as we pass through a gate on a sensitized path. The overall output deviation $\Delta^*_{OUT}$ on the output net is calculated as:

$$\Delta^*_{OUT} = \max_{i \leq j \leq K} \{\Delta_{OUT,j}\}. \hspace{1cm} (2.2)$$

**Case (ii):** Suppose $L$ input ports ($L > 1$), indexed $1, 2, \ldots, L$, are required to
make a transition in order for the gate output to change. Let $d_{\text{max}}^* = \max_{1 \leq j \leq L} \{d_j\}$.

The output deviation for the gate in this case is defined as:

$$\Delta_{\text{OUT}}^* = \prod_{i=1}^{L} P_{\text{OUT},i} \cdot (1 - d_{\text{max}}^*). \quad (2.3)$$

Note that $\Delta_{\text{OUT}}^* \leq P_{\text{OUT},i}, 1 \leq i \leq L$, since $0 \leq d_{\text{max}}^* \leq 1$. Therefore, we conclude that the probability of getting the expected transition on a net either decreases or remains the same as we pass through a logic gate. In other words, the deviation is monotonically non-decreasing along a sensitized path.

**Example:** Fig. 2.4 shows signal-transition probabilities and their propagation for a simple circuit. The test stimuli and the expected fault-free transitions on each net are shown in dark boxes. The calculated STPs are shown in angled brackets ($\langle \ldots \rangle$). The DDPMs of the gates (OR, AND, XOR, and INV) used in this circuit are given in Tables 2.1 and 2.2. The entries in both tables are chosen arbitrarily.
Table 2.2: Example DDPM for AND, XOR, INV

<table>
<thead>
<tr>
<th></th>
<th>Initial Input State</th>
<th>Inputs</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>prob</td>
<td>00</td>
<td>01</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>Inputs</td>
<td>IN0</td>
<td>0.2</td>
<td>0.3</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>IN1</td>
<td>0</td>
<td>0.2</td>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td>XOR</td>
<td>prob</td>
<td>00</td>
<td>01</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>Inputs</td>
<td>IN0</td>
<td>0.3</td>
<td>0.4</td>
<td>0.1</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>IN1</td>
<td>0.3</td>
<td>0.4</td>
<td>0.2</td>
<td>0.4</td>
</tr>
<tr>
<td>INV</td>
<td>prob</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inputs</td>
<td>IN0</td>
<td>0.2</td>
<td>0.2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the following example, the deviations are calculated as follows, based on the rules mentioned above for the example circuit in Fig. 2.4:

- Net E: There is no output change, which implies that E has the STP $\langle 1, 0, 0, 0 \rangle$.

- Net F: The output changes due to IN1 (net D) of XOR. There is a delay-defect probability of 0.4. It implies that with a probability of 0.4, the output will stay at LOW value, i.e., the STP for net F is $\langle 0.4, 0.6, 0, 0 \rangle$.

- Net G: Output changes due to IN0 (net D) of INV, i.e., the STP for net G is $\langle 0.2, 0.8, 0, 0 \rangle$.

- Net H: Output changes due to IN1 (net F) of OR.
  - If IN1 stays at LOW, output does not change. Therefore, the STP for net H is $0.4 \odot \langle 1, 0, 0, 0 \rangle$, where $\odot$ denotes the dot product;
  - If IN1 goes to HIGH, output changes with a delay defect probability of 0.2, i.e., the STP for net H is $0.6 \odot \langle 0.2, 0.8, 0, 0 \rangle$;
  - Combining all the above cases, the STP for net H is $\langle 0.52, 0.48, 0, 0 \rangle$. 

58
• Net J: Output changes due to both IN0 (net F) and IN1 (net G) of AND (both required).

  ◦ If both stay at LOW, output does not change, which implies that J has the STP $0.4 \odot 0.2(1, 0, 0, 0)$;
  ◦ If one of them stays at LOW, output does not change, i.e., the STP for net J is $0.4 \odot 0.8(1, 0, 0, 0) + 0.6 \odot 0.2(1, 0, 0, 0)$;
  ◦ If both go to HIGH, the output changes with a delay-defect probability. Since both inputs change, we use the maximum delay defect probability, i.e., the STP for net J is $0.6 \odot 0.8 \odot (0.3, 0.7, 0, 0)$;
  ◦ Combining all the above cases, the STP for net J is $(0.664, 0.336, 0, 0)$.

• Net Q1: The output changes due to only one of the inputs of OR. We need to calculate the deviation for both cases and select the one that causes maximum deviation at the output (Q1).

  ◦ For IN0 (net H) of OR:
    ◦ If IN0 stays at LOW, the output does not change, i.e., the STP for net Q1 is $0.52 \odot (1, 0, 0, 0)$;
    ◦ If IN0 goes to HIGH, the output changes with a delay-defect probability, i.e., the STP for net Q1 is $0.48 \odot (0.5, 0.5, 0, 0)$;
    ◦ Combining all the above cases, the STP for net Q1 is $(0.76, 0.24, 0, 0)$.
  ◦ For IN1 (net J) of OR:
    ◦ If IN1 stays at LOW, the output does not change, i.e., the STP for net Q1 is $0.664 \odot (1, 0, 0, 0)$;
If IN1 goes to HIGH, the output changes with a delay-defect probability, i.e., the STP for net Q1 is 0.336 \(\diamond\) \(0.2, 0.8, 0, 0\);

- Combining all the above cases, the STP for net Q1 is \(0.7312, 0.2688, 0, 0\).

- Since IN0 provided the higher deviation, we finally conclude that the STP for net Q1 is \(0.76, 0.24, 0, 0\).

Hence the deviation on Q1 is 0.76.

### 2.1.3 Implementation of Algorithm for Propagating Signal-Transition Probabilities

A depth-first procedure is used to compute signal-transition probabilities for large circuits. When we use a depth-first algorithm, only the nets that are required to find the output deviation on a specific observation point are processed. In this way, a smaller number of gate pointer stacking is required compared to the alternative of simulating the deviations starting from INs and tracing forward.

We first assign signal-transition probabilities to all INs. Then, we start from the observation points (outputs) and backtrace until we find a Processed Net (PN). A PN has all the signal transition probabilities assigned. The pseudocode for the algorithm is given in Fig. 2.5.

If the number of test patterns is \(N_s\) and the number of nets in the circuit is \(N_n\), the worst-case time-complexity of the algorithm is \(O(N_s \cdot N_n)\). However, since the calculation for each pattern is independent of other patterns (we assume full-scan designs in this chapter), the algorithm can easily be made multi-threaded. In this case, if the number of threads is \(T\), the complexity of the algorithm is reduced to
Procedure: Propagate Probability ($t_i$)
1: reset all signal-transition probabilities
2: read pattern $t_i$
3: assign signal-transition probabilities to $INs$
4: reset stack
5: for all observation points $PO_j$, $j = 1, 2, \ldots$ do
6: if $PO_j$ is processed then
7: go to next $PO_j$
8: end if
9: trace backward until a processed net is found
10: add unprocessed gates on the traced path to the stack
11: for all $G$ = gate in stack do
12: find signal-transition probabilities of the output net of $G$
13: remove $G$ from the stack
14: end for
15: find signal-transition probabilities of $PO_j$
16: end for

Figure 2.5: Signal transition probability propagation algorithm for calculating output deviations.

$O(\frac{N_s \cdot N_n}{T})$.

2.1.4 Pattern-Selection Method

In this subsection, we describe how to use output deviations to select high-quality patterns from an $n$-detect transition-fault pattern set. The number of test patterns to be selected is a user input, e.g., $S$. The parameter $S$ can be set to the number of 1-detect timing-unaware patterns, the number of timing-aware patterns, or any other value that fits the user’s test budget.

In our pattern-selection method, we target topological coverage as well as long-path coverage. As a result, we attempt to select patterns that sensitize a wide range of distinct long paths. In this process, we also discard low-quality patterns in order to find a small set of high-quality patterns.
For each test observation point $PO_j$, we keep a list of $N_p$ most effective patterns in $EFF_j$ (Fig. 2.6, lines 1-3). The patterns in $EFF_j$ are the best unique-pattern candidates for exciting a long path through $PO_j$. During deviation computation, no pattern ($t_i$) is added to $EFF_j$ if the output deviation at $PO_j$ is smaller than a limit ratio ($D_{LIMIT}$) of the maximum instantaneous output deviation (Fig. 2.6, line 10). ($D_{LIMIT}$) can be used to discard low quality patterns. If the output deviation is larger than this limit, we first check whether we have added a pattern to $EFF_j$ with the same deviation (Fig. 2.6, line 11). It is unlikely that two different patterns will create the same output deviation on the same output $PO_j$ while exciting different non-redundant paths. Since we want a higher topological path-coverage, we skip these cases (Fig. 2.6, line 11). Although this assumption may not necessarily be true, we assume for the sake of completeness that it holds for most cases. If we observed a unique deviation on $PO_j$, we first check whether $EFF_j$ is full (already includes $N_p$ patterns); see Fig. 2.6, line 12. Pattern $t_i$ is added to $EFF_j$ along with its deviation if $EFF_j$ is not full or if $t_i$ has a larger deviation than the minimum deviation stored in $EFF_j$ (Fig. 2.6, lines 12-17). The effectiveness of a pattern is measured by the number of occurrences of this pattern in $EFF_j$ for all values of $j$. For instance, if at the end of deviation computation, pattern $A$ was included in the $EFF$ list for 10 observation points, and pattern $B$ was listed in the $EFF$ list for 8 observation points, we conclude that pattern $A$ is more effective than pattern $B$.

Once the deviation computation is completed, the list of pattern effectiveness is generated and the final pattern filtering and selection is carried out (Fig. 2.7). First, pattern effectiveness is generated (Fig. 2.7, lines 1-9). Since $Max_Dev$ is updated on the fly, we may miss some low quality patterns. As a result, we need to filter by
**Procedure:** Compute Deviations \((t_0, ..., t_{N_s}, N_p)\)

1: **for all** observation point \(PO_j, j = 1, 2,.. \) **do**
2: create list \(EFF_j[N_p]\)
3: **end for**
4: Max\_Dev = 0;
5: **for all** test pattern \(t_i, i = 1, 2, ..., N_s \) **do**
6: Propagate Probability\((t_i)\);
7: **for all** observation point \(PO_j, j = 1, 2, .. \) **do**
8: \(Dev = \) deviation of \(PO_j\);
9: if \(Dev > \) Max\_Dev then Max\_Dev = \(Dev\);
10: if \(Dev > D_{LIMIT} \cdot \) Max\_Dev then
11: if \(EFF_j\) includes Dev then Next observation point;
12: if \(EFF_j\) is not full then
13: add \(t_i\) and \(Dev\) to \(EFF_j\);
14: else if \(Dev > \) min\(\)\((EFF_j)\) then
15: remove \(min\)\((EFF_j)\);
16: add \(t_i\) and \(Dev\) to \(EFF_j\);
17: **end if**
18: **end if**
19: **end for**
20: **end for**

**Figure 2.6:** Deviation-computation algorithm for pattern selection.

Max\_Dev \((D_{LIMIT})\) again to discard low quality patterns from the final pattern list (Fig. 2.7, line 5). Setting \(D_{LIMIT}\) to a high value may result in discarding most of the patterns, leaving only the best few patterns. Depending on \(D_{LIMIT}\), the number of remaining patterns can be less than \(S\). In the next stage, the patterns are re-sorted by their effectiveness (Fig. 2.7, line 10). Finally, until the selected pattern number reaches \(S\) or all patterns are selected, the top patterns are selected (Fig. 2.7, line 11).

The computational complexity of the selection algorithm is \(O(N_sp)\), where \(N_s\) is the number of test patterns and \(p\) is the number of observation points. This procedure is very fast since it only includes two nested for loops and a simple list-item existence check.
Procedure: Select Patterns \( D_S(t_0, \ldots, t_{N_s}, S, D_{LIMIT}) \)

1: list \( D[N_s] \);
2: init \( D \) to all 0s;
3: for all test pattern \( t_i, i = 1, 2, \ldots, N_s \) do
   4:   for all observation point \( PO_j, j = 1, 2, \ldots \) do
      5:     if \( EFF_j \) includes \( t_i \) and deviation of \( t_i > D_{LIMIT} \) \( \cdot \) \( \text{Max}_\text{Dev} \) then
         6:         increment \( D[i] \);
      7:     end if
   8:   end for
9: end for
10: sort \( D \) by values;
11: \( D_S = \) select top \( S \) patterns;
12: return \( D_S \);

Figure 2.7: Pattern selection algorithm.

2.2 Experimental Results

In this section, we present experimental results obtained for the IWLS benchmark circuits. We first describe how we obtained DDPs for the logic gates in these benchmarks (Section 2.2.1). Next, we provide details for the benchmark circuits. Then, we provide details for the experimental setup (Section 2.2.3). Before presenting the simulation results (Sections 2.2.5 and 2.2.6), we provide a summary of the dynamic-timing simulation method proposed in [60], which will be compared with our proposed deviation-based method (Section 2.2.4).

2.2.1 Finding Gate-Delay Defect Probabilities

In order to determine the gate-DDPs, we ran 200 HSpice Monte Carlo (MC) simulations on each gate, for all possible input signal transitions, using 45 nm process-technology BSIM4 predictive transistor models. We verified that 200 MC simulations
are sufficient for generating DDPMs. Fig. 2.8 shows the relative change in DDPM entries for various gates as the number of MC simulations increase. The numbers are normalized by the values obtained from 50 MC simulations. As seen, there is very little change in the DDPM entries well before we reach 200 MC simulations. Therefore, we conclude that running more MC simulations will not lead to any significant difference in the DDPM entries. Similar results were obtained for other library cells.

For each gate type, we simulated the schematic under various loading capacitance and input slew rate conditions to account for spatial correlations. Next, we used interpolation to find the mean and standard deviation of gate delays for individual gate instances. We have seen that interpolation is possible and it is accurate. For instance, the mean delay value approximately linearly changes with the load capacitance, and further accuracy can be achieved by using third-order polynomial curve fitting.

MC simulations were carried out using the following realistic process-variation parameters (obtained from a current VDSM technology from industry) for a Gaussian distribution: Transistor gate length $L$: $3\sigma = 10\%$, threshold voltage $V_{TH}$: $3\sigma = 30\%$, and gate-oxide thickness $t_{OX}$: $3\sigma = 3\%$. For each configuration, MC simulations were performed for each possible input transition. For each gate, the probability that the transition-delay value is greater than $T_{CRT} = \mu + \tau$ constitutes the DDP value for the respective input transition (where $\mu$ refers to the nominal delay value). The parameter $\tau$ is selected in such a way that all the gate instances have at least one non-zero DDPM entry. Note that $\tau$ is the minimum of all MAX delays (among all gates) for each benchmark. This corresponds to the maximum delay of an inverter driving a single-inverter load. Note that selecting too large a value for $\tau$ may cause
many DDPMs to have all-zero entries, simply because the gate would never have a
delay larger than $\mu + \tau$. As an alternative, $\tau$ can be set to the $\text{MAX}$ delay specified
by an STA tool that does not consider process variation effects.

### 2.2.2 Benchmarks

In this section, we present the details of the IWLS 2005 benchmark circuits. We do
not consider the ISCAS benchmarks because these circuits are small and it is easier
for an ATPG tool to excite all long paths with a small number of patterns.

IWLS benchmark circuit statistics are shown in Table 2.3. As seen, IWLS bench-
marks represent a wide range of application areas, including memory controllers
and microprocessors. Note that IWLS benchmarks are provided in the Verilog
RTL format, and the statistics given in Table 2.3 may slightly change depending
on the synthesis tool and synthesis optimization options. For our experiments,
we selected a subset of the IWLS benchmarks: systemcaes, usb_funct, ac97_ctrl,
aes_core, pci_bridge32, wb_conmax, and ethernet. The largest benchmarks require a
prohibitive amount of computing resources for the collection of simulation data for
pattern quality evaluation (sensitized paths and coverage ramp-up), therefore we do
not report results for them.

### 2.2.3 Experimental Setup

All experiments were performed on a pool of state-of-the-art servers with at least
eight processors available at all times, 16GB of memory, and running Linux. The
program to compute output deviations was implemented using C++. A commercial
tool was used to perform Verilog netlist synthesis and scan insertion for the IWLS
<table>
<thead>
<tr>
<th>Benchmark</th>
<th># I/O</th>
<th># gates</th>
<th># flip-flops</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>wb_dma</td>
<td>942</td>
<td>7,619</td>
<td>563</td>
<td>WISHBONE DMA/Bridge IP Core</td>
</tr>
<tr>
<td>tv80</td>
<td>103</td>
<td>13,326</td>
<td>359</td>
<td>TV80 8-Bit Microprocessor Core</td>
</tr>
<tr>
<td>systemcaes</td>
<td>387</td>
<td>17,817</td>
<td>670</td>
<td>SystemC AES</td>
</tr>
<tr>
<td>mem_ctrl</td>
<td>208</td>
<td>22,015</td>
<td>1,083</td>
<td>WISHBONE Memory Controller</td>
</tr>
<tr>
<td>usb_func</td>
<td>167</td>
<td>25,531</td>
<td>1,746</td>
<td>USB function core</td>
</tr>
<tr>
<td>ac97_ctrl</td>
<td>115</td>
<td>28,083</td>
<td>2,199</td>
<td>WISHBONE AC 97 Controller</td>
</tr>
<tr>
<td>aes_core</td>
<td>267</td>
<td>29,186</td>
<td>530</td>
<td>AES Cipher</td>
</tr>
<tr>
<td>dma</td>
<td>120</td>
<td>41,026</td>
<td>2,189</td>
<td>Direct Memory Access (DMA) Controller</td>
</tr>
<tr>
<td>pci_bridge32</td>
<td>361</td>
<td>43,907</td>
<td>3,359</td>
<td>PCI Interface</td>
</tr>
<tr>
<td>wb_conmax</td>
<td>388</td>
<td>59,484</td>
<td>770</td>
<td>WISHBONE Conmax IP Core</td>
</tr>
<tr>
<td>ethernet</td>
<td>45</td>
<td>153,948</td>
<td>10,544</td>
<td>Ethernet IP core</td>
</tr>
<tr>
<td>netcard</td>
<td>186</td>
<td>1,356,533</td>
<td>97,831</td>
<td>Network Card Controller</td>
</tr>
<tr>
<td>leon3mp</td>
<td>2,546</td>
<td>1,452,517</td>
<td>108,839</td>
<td>32-bit processor compliant with SPARC V8 arch.</td>
</tr>
</tbody>
</table>
benchmark circuits. We used a commercial ATPG tool to generate $n$-detect TDF test patterns and timing-aware TDF patterns for these circuits. The ATPG tool was forced to generate launch-on-capture (LOC) transition fault patterns. The primary input change during capture cycles and the observation of primary outputs was prevented in order to simulate realistic test environments. The path delays were calculated using an in-house dynamic path-timing simulator. All simulations were run in parallel on eight processors.

### 2.2.4 Dynamic Timing Simulation

In this chapter, we compare our proposed method with the one presented in [60]. Two methods for estimating the path delays for a given test vector are proposed in [60]. In particular, a path-based and a cone-based delay estimation scheme were provided. The estimated delays are called “metrics” associated with the test vectors. Two different delay models are used during the estimation process. (i) **Unit delay model**: Each gate has a unit delay, no spatial correlations are considered. (ii) **Differential delay model**: The gate type and the number of fanouts are considered.

For the path-based scheme, it is assumed that a set of critical paths is given. Non-robustly excited critical paths and their corresponding delays are found using either the unit or the differential delay model. The largest delay caused by the test pattern is assigned as the “metric” for the pattern. If there is no non-robust transition on a gate, zero delay is assumed on that gate. Because of the non-robust restriction, delay accuracy is lost. For the cone-based method, there are no non-robust restrictions, but delays are calculated for small cones of logic, and then the delays of cones are added to approximate the overall delay. Once all test vectors are associated with a
“metric”, the patterns are ordered by the “metric” and the top 1/3 of the patterns are selected.

In order to compare our work to the dynamic-timing simulation based method, we implemented the cone-based scheme using the differential delay model. Instead of the number of fanouts, we used capacitive loading of fanout gates to update the delay associated with each gate instance. The capacitive loading of gate ports are found by running HSpice simulations on transistor-level gate models. Note that in the absence of layouts, this data does not include layout-extracted data such as resistances and capacitances. Once the “metrics” are computed, we selected the top 1/3 of the patterns as proposed by Lee et al. [60].

2.2.5 Correlation Between Output Deviations and Path Lengths

We ran correlation analysis to determine the relationship between output deviations and sensitized path lengths. For each benchmark, we calculated output deviations and path lengths for n-detect transition-fault test-patterns ($n = 1, 3, 5, 8, 10$). We simulated these patterns using the in-house dynamic timing simulator and determined the signal delays at the observation points of the benchmarks. Next, we used Matlab to compute the Kendall’s correlation coefficients [62] for each pattern set. The Kendall’s correlation coefficient is a measure between 0 and 1, where 0 indicates no correlation and 1 indicates perfect correlation. Table 2.4 shows the average correlation coefficients for the patterns in a 1-detect test set of the IWLS’2005 benchmarks [43]. The results for other values of $n$ are similar. The minimum and maximum values of the correlation coefficients are also given. As seen in Table 2.4, there is a strong positive correlation (close to the perfect correlation measure of 1)
Table 2.4: Kendall’s coefficients for evaluating the correlation of path lengths to output deviations

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>(Ave, Min, Max)</th>
<th>Benchmark</th>
<th>(Ave, Min, Max)</th>
</tr>
</thead>
<tbody>
<tr>
<td>systemcaes</td>
<td>(0.96, 0.82, 1)</td>
<td>pci_bridge32</td>
<td>(0.93, 0.85, 0.99)</td>
</tr>
<tr>
<td>usb_funct</td>
<td>(0.97, 0.85, 0.99)</td>
<td>wb_conmax</td>
<td>(0.93, 0.89, 0.98)</td>
</tr>
<tr>
<td>ac97_ctrl</td>
<td>(0.98, 0.93, 1)</td>
<td>ethernet</td>
<td>(0.89, 0.77, 0.95)</td>
</tr>
<tr>
<td>aes_core</td>
<td>(0.97, 0.9, 0.99)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

between output deviations and path lengths. Thus, the method of output deviations is a promising metric for evaluating the capability of transition delay-test patterns to sensitize long paths.

It can be argued that instead of output deviations, a dynamic timing simulator can be used to obtain high correlation to path lengths. However, the method based on output deviations is flexible and general, and it can be used during pattern selection to account for process variations and many physical defects. Dynamic timing simulation can only provide variability-unaware timing information. The method of output deviations is also expected to reveal unique problematic paths that may be hidden from dynamic timing analysis.

2.2.6 Pattern-Selection Results

In this section, we present the pattern-selection results for the proposed deviation-based method (dev) and the dynamic-timing simulation based method (dts) [60] for a production test environment with pattern-count limits.

We first generated 5-detect (n5), 8-detect (n8), and timing-aware (ta) transition-test patterns for each benchmark using a commercial ATPG tool. The raw CPU time data is given in Table 2.5. Next, dev and dts are used to select patterns
from these base pattern sets. The motivation for also using timing-aware patterns as a base pattern-set lies in our observation that the pattern counts resulting from timing-aware ATPG for large industrial circuits are often prohibitively high. Test-set truncation is therefore necessary in practice. In order to obtain the same TDF coverage as 1-detect patterns, we ran top-off timing-unaware ATPG over the selected patterns.

The number of patterns generated by the commercial ATPG tool is given in Table 2.6. When \( dts \) is used as the pattern selection scheme, the top 1/3 patterns (ranked on the basis of the “metric”) of the base patterns (n5, n8, and ta) are selected as in [60]; see Table 2.7. The number of patterns after top-off ATPG for TDFs is given in parentheses.

For \( \text{dev} \), we used a range of \( D_{\text{LIMIT}} \) values for pattern selection. The results for \( D_{\text{LIMIT}} = 0.6 \) and \( D_{\text{LIMIT}} = 0.8 \) are shown in Table 2.8. We see that an increase in \( D_{\text{LIMIT}} \) resulted in a lower pattern count in the selected pattern set, but increased the number of top-off ATPG patterns. We observe that for most benchmarks, the selected pattern count is significantly smaller than the number of base timing-aware ATPG patterns, and even smaller than the number of patterns selected by \( dts \). For the rest of the experimental results presented in this section for \( \text{dev} \), we used the patterns selected with \( D_{\text{LIMIT}} = 0.8 \).

The CPU time required by timing-aware ATPG is an important concern, especially for large industrial designs under time-to-market constraints. We have seen that for large industrial circuits, timing-unaware TDF ATPG itself can take a couple of days to complete. Therefore, the CPU time for timing-aware ATPG can be prohibitive, as shown in Fig. 1.28 for benchmark circuits. We evaluated the total CPU
time used by dev and dts and compared it to the CPU time used by the base timing-aware ATPG. The results are shown in Fig. 2.9. We find that, even with CPU time for top-off ATPG, for \( n \)-detect pattern sets, dev consistently has lower CPU time compared to both base timing-aware ATPG and dts. For the ta pattern group, the CPU time used for pattern selection and fault-grading is significantly smaller than the base timing-aware ATPG run time.

In order to evaluate the effectiveness of selected patterns in terms of long-path sensitization, we ran timing simulations for the selected patterns and found the number of sensitized distinct long paths. Two paths are assumed to be distinct if there is at least one net that is not shared by them. A path is assumed to be a long path if the corresponding delay is higher than the specified long path limit (LPL). We ran the analysis for a range of LPL values, starting from 50% of the clock period to 90% of the clock period. The results for 90% long-path limit is shown in Fig. 2.10. The results for other values of LPL follow the same trend. The contribution of the selected patterns and the top-off patterns is shown in different colors. Note that the number of patterns for the base timing-aware ATPG is much larger than any of the pattern selection methods. Therefore, we trimmed the base timing-aware ATPG patterns and selected the first \( P \) patterns (ranked by the ATPG tool) as a baseline where \( P \) is the maximum of the number of selected patterns (including top-off patterns) either by dev or dts. The results for the trimmed timing-aware ATPG patterns are shown as horizontal lines. As seen in Fig. 2.10, dev consistently excites more long paths than dts. We also note that the proposed method sensitizes more long paths than the timing-aware ATPG baseline with \( P \) patterns for three circuits—ac97_crl, pci_bridge32, and wb_conmax. It sensitizes nearly the same number of long
paths for systemcaes and ethernet. Recall that in all cases, the CPU time for timing-aware ATPG is much higher (Fig. 2.9). The results for the full base ATPG pattern sets are shown in Figs. 2.11-2.17. As seen, although decreasing the pattern dropping threshold to 0.3 changes the selected pattern count, the change is small and pattern quality is preserved.
Figure 2.8: Change in DDPM values as a function of the number of MC simulations, relative to the base case of 50 MC simulations. (a) OR gate, (b) AND gate, (c) INV, BUF, and XOR gates.
Table 2.5: ATPG tool CPU time (in seconds) used to generate initial pattern sets

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>n1</th>
<th>n3</th>
<th>n5</th>
<th>n8</th>
<th>ta</th>
</tr>
</thead>
<tbody>
<tr>
<td>systemcaes</td>
<td>11.20</td>
<td>37.90</td>
<td>63.90</td>
<td>100.10</td>
<td>115.60</td>
</tr>
<tr>
<td>usb_funct</td>
<td>3.30</td>
<td>17.00</td>
<td>31.40</td>
<td>52.00</td>
<td>207.80</td>
</tr>
<tr>
<td>ac97_ctrl</td>
<td>6.10</td>
<td>14.30</td>
<td>26.70</td>
<td>44.50</td>
<td>111.40</td>
</tr>
<tr>
<td>aes_core</td>
<td>37.10</td>
<td>46.70</td>
<td>74.70</td>
<td>117.50</td>
<td>1,892.90</td>
</tr>
<tr>
<td>pci_bridge32</td>
<td>7.70</td>
<td>34.50</td>
<td>73.30</td>
<td>131.10</td>
<td>493.80</td>
</tr>
<tr>
<td>wb_conmax</td>
<td>16.70</td>
<td>106.70</td>
<td>237.70</td>
<td>431.30</td>
<td>827.00</td>
</tr>
<tr>
<td>ethernet</td>
<td>128.30</td>
<td>444.30</td>
<td>953.60</td>
<td>1,695.80</td>
<td>5,589.50</td>
</tr>
<tr>
<td>vga_led</td>
<td>376.00</td>
<td>1,062.00</td>
<td>2,277.30</td>
<td>4,103.30</td>
<td>10,332.60</td>
</tr>
<tr>
<td>netcard</td>
<td>9,605.70</td>
<td>38,944.90</td>
<td>85,202.10</td>
<td>Not run</td>
<td>2,008,786.10</td>
</tr>
<tr>
<td>leon3mp</td>
<td>13,476.00</td>
<td>44,723.10</td>
<td>88,611.20</td>
<td>Not run</td>
<td>41,451.90</td>
</tr>
</tbody>
</table>

Table 2.6: Number of test patterns generated by the commercial ATPG tool for various n-detect TDF and timing-aware ATPG (ta)

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>n5</th>
<th>n8</th>
<th>ta</th>
</tr>
</thead>
<tbody>
<tr>
<td>systemcaes</td>
<td>1939</td>
<td>2821</td>
<td>2997</td>
</tr>
<tr>
<td>usb_funct</td>
<td>3802</td>
<td>5797</td>
<td>3683</td>
</tr>
<tr>
<td>ac97_ctrl</td>
<td>1907</td>
<td>2866</td>
<td>1643</td>
</tr>
<tr>
<td>aes_core</td>
<td>2734</td>
<td>3995</td>
<td>8277</td>
</tr>
<tr>
<td>pci_bridge32</td>
<td>4789</td>
<td>7383</td>
<td>4571</td>
</tr>
<tr>
<td>wb_conmax</td>
<td>21230</td>
<td>32189</td>
<td>6418</td>
</tr>
<tr>
<td>ethernet</td>
<td>25221</td>
<td>36855</td>
<td>13544</td>
</tr>
</tbody>
</table>

Table 2.7: Number of test patterns (one-third of base set) selected by the dynamic-timing simulation based scheme (dts).

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>n5*</th>
<th>n8*</th>
<th>ta*</th>
</tr>
</thead>
<tbody>
<tr>
<td>systemcaes</td>
<td>646 (784)</td>
<td>940 (991)</td>
<td>999 (1151)</td>
</tr>
<tr>
<td>usb_funct</td>
<td>1267 (1474)</td>
<td>1932 (2010)</td>
<td>1226 (1695)</td>
</tr>
<tr>
<td>ac97_ctrl</td>
<td>635 (815)</td>
<td>955 (1019)</td>
<td>551 (783)</td>
</tr>
<tr>
<td>aes_core</td>
<td>911 (1097)</td>
<td>1331 (1384)</td>
<td>2759 (2766)</td>
</tr>
<tr>
<td>pci_bridge32</td>
<td>1596 (1951)</td>
<td>2461 (2693)</td>
<td>1507 (2069)</td>
</tr>
<tr>
<td>wb_conmax</td>
<td>7076 (7831)</td>
<td>10729 (10961)</td>
<td>2142 (5713)</td>
</tr>
<tr>
<td>ethernet</td>
<td>8407 (10213)</td>
<td>12284 (12877)</td>
<td>4514 (9708)</td>
</tr>
</tbody>
</table>

*Numbers in parenthesis refer to the pattern count after top-off timing-unaware ATPG.
Table 2.8: Number of test patterns selected by the deviation-based scheme (dev) when $D_{LIMIT} = 0.6$ and $D_{LIMIT} = 0.8$.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>$D_{LIMIT} = 0.6^*$</th>
<th></th>
<th></th>
<th>$D_{LIMIT} = 0.8^*$</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n5</td>
<td>n8</td>
<td>ta</td>
<td>n5</td>
<td>n8</td>
<td>ta</td>
</tr>
<tr>
<td>systemcaes</td>
<td>974 (1054)</td>
<td>1135 (1210)</td>
<td>1245 (1417)</td>
<td>938 (1027)</td>
<td>1097 (1174)</td>
<td>1205 (1374)</td>
</tr>
<tr>
<td>usb_funct</td>
<td>1171 (1360)</td>
<td>1330 (1494)</td>
<td>1298 (1751)</td>
<td>659 (1110)</td>
<td>742 (1149)</td>
<td>744 (1390)</td>
</tr>
<tr>
<td>ac97_ctrl</td>
<td>614 (740)</td>
<td>699 (805)</td>
<td>575 (801)</td>
<td>281 (589)</td>
<td>246 (582)</td>
<td>63 (544)</td>
</tr>
<tr>
<td>aes_core</td>
<td>1104 (1214)</td>
<td>1232 (1309)</td>
<td>1379 (1479)</td>
<td>1060 (1182)</td>
<td>1179 (1277)</td>
<td>1294 (1413)</td>
</tr>
<tr>
<td>pci_bridge32</td>
<td>1069 (1525)</td>
<td>1287 (1683)</td>
<td>1432 (2005)</td>
<td>408 (1319)</td>
<td>459 (1341)</td>
<td>504 (1452)</td>
</tr>
<tr>
<td>wb_conmax</td>
<td>2434 (5588)</td>
<td>2572 (5608)</td>
<td>1949 (5648)</td>
<td>2091 (5575)</td>
<td>2275 (5535)</td>
<td>1525 (5651)</td>
</tr>
<tr>
<td>ethernet</td>
<td>11967 (13740)</td>
<td>13448 (14606)</td>
<td>8877 (11472)</td>
<td>11479 (13453)</td>
<td>12911 (14304)</td>
<td>8493 (11351)</td>
</tr>
</tbody>
</table>

*Numbers in parenthesis refer to the pattern count after top-off timing-unaware ATPG.
Figure 2.9: Comparison of CPU time for dev and dts. The CPU times used for each step is shown in a different color. The values are normalized by the timing-aware ATPG CPU time for each benchmark. The normalization point (timing-aware ATPG CPU time) is shown as a red line.
Figure 2.10: The number of sensitized distinct long paths for dev and dts (LPL=90%): (a) systemcaes; (b) usb_funct; (c) ac97_ctrl; (d) aes_core; (e) pci_bridge32; (f) wb_conmax; (g) ethernet. The number of sensitized distinct long paths for the trimmed timing-aware ATPG patterns is shown as a horizontal line.
**Figure 2.11**: The number of sensitized distinct long paths by dev and dts for benchmark systemcaes compared to full base pattern sets (LPL=90%): (a) $D_{LIMIT} = 0.8$; (b) $D_{LIMIT} = 0.3$; (c) Pattern count comparison table.
Figure 2.12: The number of sensitized distinct long paths by dev and dts for benchmark usb_func compared to full base pattern sets (LPL=90%): (a) $D_{LIMIT} = 0.8$; (b) $D_{LIMIT} = 0.3$; (c) Pattern count comparison table.
Figure 2.13: The number of sensitized distinct long paths by dev and dts for benchmark ac97_ctrl compared to full base pattern sets (LPL=90%): (a) $D_{LIMIT} = 0.8$; (b) $D_{LIMIT} = 0.3$; (c) Pattern count comparison table.
Figure 2.14: The number of sensitized distinct long paths by \texttt{dev} and \texttt{dts} for benchmark \texttt{aes_core} compared to full base pattern sets (LPL=90%): (a) $D_{\text{LIMIT}} = 0.8$; (b) $D_{\text{LIMIT}} = 0.3$; (c) Pattern count comparison table.
Figure 2.15: The number of sensitized distinct long paths by dev and dts for benchmark pci_bridge32 compared to full base pattern sets (LPL=90%): (a) $D_{LIMIT} = 0.8$; (b) $D_{LIMIT} = 0.3$; (c) Pattern count comparison table.
Figure 2.16: The number of sensitized distinct long paths by \texttt{dev} and \texttt{dts} for benchmark \texttt{wb_conmax} compared to full base pattern sets (LPL=90%): (a) $D_{LIMIT} = 0.8$; (b) $D_{LIMIT} = 0.3$; (c) Pattern count comparison table.
**Figure 2.17**: The number of sensitized distinct long paths by dev and dts for benchmark ethernet compared to full base pattern sets (LPL=90%): (a) $D_{LIMIT} = 0.8$; (b) $D_{LIMIT} = 0.3$; (c) Pattern count comparison table.
To further evaluate the long path sensitization capability of the selected patterns, we determined the delay distribution of the excited distinct long paths. We set LPL to 80% of the rated clock period. The results are shown in Fig. 2.18-2.19. The results shown in these figures are for the base 8-detect pattern set (n8(base)), dts-based selected patterns and top-off patterns (n8(dts)), and dev-based selected patterns with \(D_{\text{LIMIT}} = 0.8\) and top-off patterns (n8(dev)). To draw appropriate conclusions, it is necessary to examine the pattern counts in Table 2.6 (column n8), Table 2.7 (column n8), and Table 2.8 (column n8 under \(D_{\text{LIMIT}} = 0.8\)). Note that the base pattern set has a much larger pattern count compared to selected pattern sets with additional top-off patterns. Furthermore, dev-based selection leads to lower pattern count than dts for all benchmarks except systemcaes and ethernet. The difference in pattern count is especially significant for pci_bridge32 and wb_conmax, where dev-based selection has almost half of the pattern count of dts. We find that dev-based selection clearly outperforms dts-based selection for most benchmarks, even with fewer patterns. For pci_bridge32 and wb_conmax, dev-based selection is almost as effective as dts-based selection even though the pattern count is only half. Another important observation is that dev-based selection successfully extracted almost all of the long path sensitizing patterns from the base pattern sets for most benchmarks.

To evaluate the fault coverage ramp-up provided by dev, dts, and timing-aware ATPG, we ran fault injection simulations. For each benchmark, we inserted 50000 delay defects on randomly chosen nets. We assumed that the additional delay introduced by the injected defects has a distribution of \(e^{-Ax}\) as used in [44] and [3], and we injected random delay defects using the method described below.

We let \(A = \frac{5}{T_{\text{CLK}}}\), where \(T_{\text{CLK}}\) corresponds to the rated clock period of the circuit
under test. The delay-defect distribution used in [44] and [3] is similar to the one shown in Fig. 2.20. However, on the y-axis, uniformly distributed random numbers are shown instead of the number of parts. We use the distribution

\[
y = e^{-\frac{5x}{TCLK}}, \quad 0 < y < 1.
\] (2.4)

If Equation (2.4) is solved to determine \( x \) (additional delay) in terms of \( y \) (un-
Figure 2.19: The distribution of sensitized distinct paths (LPL=80%) for base 8-detect patterns, dts-based selection, and dev-based selection: a) pci_bridge32; (b) wb_conmax; (c) ethernet.

formly distributed random number), the following equation results:

\[ x = -\ln y \cdot \frac{T_{CLK}}{5}. \]  

(2.5)

The graph corresponding to Equations (2.4) and (2.5) is shown in Fig. 2.20 for \( T_{CLK} = 100\, ps \). A total of 50000 uniformly distributed random numbers were generated and corresponding delay defects were injected in the circuit under test. As seen in Fig. 2.20, 70% of the injected delay defects are less than 20% of the clock
Thus, our defect-injection mechanism injected more small-delay defects than gross-delay defects, as is the case for VDSM technology [44].

The number of detected faults for all the benchmarks is presented as Venn diagrams in Fig. 2.21. Although it is only expected that dev patterns will not catch all detectable faults, we find that, for most cases, dev missed less faults compared to both dts and timing-aware ATPG.

Early detection of defects is also important in an abort-on-first-fail methodology, and it can save considerable test time. Furthermore, if the test-time budget is limited, whereby truncation is necessary and only a small portion of the patterns can be used for production test, it is important to apply the most effective patterns before less effective ones. Fig. 2.22-2.25 show how the fault coverage increases with the number of patterns for all different benchmark circuits. Each plot in these figures shows results for the base timing-aware ATPG patterns (ta(base)), the patterns selected or sorted by dev (n8(dev) and ta(dev)), and the pattern selected and sorted by dts (n8(dts) and ta(dts)). We find that the coverage rises more steeply for the proposed method (dev) compared to both dts and the base timing-aware ATPG.

![Figure 2.20: The distribution of injected delay defects.](image)
patterns.

We also report results on long-path coverage and defect detection for the full timing-aware test sets, i.e., without truncation. The results are shown in Figs. 2.26-2.29. As expected, the full timing-aware test sets provide higher coverage, but these test sets are much longer in length. For several benchmarks, comparable or even better coverage is achieved using the proposed method with much smaller test sets. For a range of smaller pattern counts, the proposed method clearly outperforms the full timing-aware test sets.

Finally, we evaluate the impact of making small perturbations to the DDPM entries. For each benchmark, we injected random variations to DDPM entries using three different maximum variation values: 10% (p:0.1), 20% (p:0.2), and 30% (p:0.3). The injected variations are uniformly distributed between $BASE - p$ and $BASE + p$, where $BASE$ shows the original DDPM entry (corresponding bar is named as p:0). The number of excited distinct long paths for the corresponding full ATPG pattern sets (n5, n8, and timing-aware) are shown as a red horizontal lines. These lines simply show the maximum achievable limit for each selected pattern set (blue bars under them). For each benchmark, the results are shown for two different long path limits: 90% of the clock period and 70% of the clock period. The results, shown in Figs. 2.34-2.40, indicate that the pattern selection results and test quality are relatively insensitive to small changes in the DDPM entries. We attribute this finding to the fact that any DDPM changes affect multiple paths in the circuits, hence their impact is amortized over the circuit and the test set. The absolute values of the output deviations are less important than the relative values for different test patterns.
Figure 2.21: The number of defects detected by the selected patterns for dev, dts, the trimmed timing-aware ATPG patterns (ta_trimmed), and full set of timing-aware ATPG patterns (ta) (ta is a super-set of all selected patterns): (a) systemcaes; (b) usb funct; (c) ac97 ctrl; (d) aes core; (e) pci bridge32; (f) wb conmax; (g) ethernet.
Figure 2.22: The fault coverage ramp-up using the selected patterns of dev and dts, and the base timing-aware ATPG patterns: systemcaes.
Figure 2.23: The fault coverage ramp-up using the selected patterns of dev and dts, and the base timing-aware ATPG patterns: (a) usb\_func; (b) ac97\_ctrl.
Figure 2.24: The fault coverage ramp-up using the selected patterns of dev and dts, and the base timing-aware ATPG patterns: (a) aes\_core; (b) pci\_bridge32.
Figure 2.25: The fault coverage ramp-up using the selected patterns of dev and dts, and the base timing-aware ATPG patterns: (a) wb_conmax; (b) ethernet.
Figure 2.26: The long-path coverage ramp-up using the selected patterns of dev and dts, and the full base timing-aware ATPG patterns (LPL=90%): (a) systemcaes; (b) usb funct.
Figure 2.27: The long-path coverage ramp-up using the selected patterns of dev and dts, and the full base timing-aware ATPG patterns (LPL=90%): (a) ac97_ctrl; (b) aes_core.
Figure 2.28: The long-path coverage ramp-up using the selected patterns of dev and dts, and the full base timing-aware ATPG patterns (LPL=90%): (a) pci_bridge32; (b) wb_conmax.
Figure 2.29: The long-path coverage ramp-up using the selected patterns of dev and dts, and the full base timing-aware ATPG patterns (LPL=90%): ethernet.

Figure 2.30: The fault coverage ramp-up using the selected patterns of dev and dts, and the full base timing-aware ATPG patterns: systemcaes.
Figure 2.31: The fault coverage ramp-up using the selected patterns of dev and dts, and the full base timing-aware ATPG patterns: (a) usb_funct; (b) ac97_ctrl.
Figure 2.32: The fault coverage ramp-up using the selected patterns of \textit{dev} and \textit{dts}, and the full \texttt{base} timing-aware ATPG patterns: (a) \texttt{aes\_core}; (b) \texttt{pci\_bridge32}. 
Figure 2.33: The fault coverage ramp-up using the selected patterns of dev and dts, and the full base timing-aware ATPG patterns: (a) wb_conmax; (b) ethernet.
Figure 2.34: The effect of DDPM perturbations to the selected pattern quality in terms of sensitized long paths for benchmark systemcaes. The perturbation limits are given in parentheses. (a) LPL=90%; (b) LPL=70%
Figure 2.35: The effect of DDPM perturbations to the selected pattern quality in terms of sensitized long paths for benchmark usb funct. The perturbation limits are given in parentheses. (a) LPL=90%; (b) LPL=70%
Figure 2.36: The effect of DDPM perturbations to the selected pattern quality in terms of sensitized long paths for benchmark ac97_ctrl. The perturbation limits are given in parentheses. (a) LPL=90%; (b) LPL=70%
Figure 2.37: The effect of DDPM perturbations to the selected pattern quality in terms of sensitized long paths for benchmark aes core. The perturbation limits are given in parentheses. 
(a) LPL=90%; (b) LPL=70%
Figure 2.38: The effect of DDPM perturbations to the selected pattern quality in terms of sensitized long paths for benchmark pci_bridge32. The perturbation limits are given in parentheses. (a) LPL=90%; (b) LPL=70%
Figure 2.39: The effect of DDPM perturbations to the selected pattern quality in terms of sensitized long paths for benchmark wb_conmax. The perturbation limits are given in parentheses. (a) LPL=90%; (b) LPL=70%
Figure 2.40: The effect of DDPM perturbations to the selected pattern quality in terms of sensitized long paths for benchmark ethernet. The perturbation limits are given in parentheses. (a) LPL=90%; (b) LPL=70%
2.3 Summary

We have presented a test-grading technique, based on output deviations, for screening small-delay defects (SDDs). We have defined the concept of output deviations for pattern-pairs and shown that it can be used as an efficient surrogate metric to model the effectiveness of transition delay-fault (TDF) patterns for SDDs. We have introduced a gate-delay defect probability measure to model delay variations for nanometer technologies. Experimental results for the IWLS’2005 benchmark circuits show that the proposed method intelligently selects the best set of patterns for SDD detection from an $n$-detect or timing-aware TDF pattern set, and it excites a larger number of long paths compared to commercial timing-aware ATPG tool. We have also shown that, the selected patterns are considerably more effective than a recently proposed method for detecting small delay defects caused by resistive shorts, resistive opens, and process variations.
Chapter 3

Layout-Aware Output Deviations

Interconnect delays are a major concern for very deep sub-micron (VDSM) process technologies [32]. The increased susceptibility of VDSM designs to process variations make interconnect delays even more important. Higher process variation and crosstalk effects have resulted in the emergence of interconnects as the major contributor to small-delay defects (SDDs) [53]. New test-generation methods are therefore needed to target SDDs caused by process variation and crosstalk on interconnects.

In this chapter, we revisit the problem of detecting SDDs by selecting the best test patterns from a repository test set for transition-delay faults, which is generated without considering process variations. The proposed method is based on the use of the output deviations metric for delay faults, as introduced in Chapter 2. The output deviation measure is once again used as a coverage-metric for SDDs. We also show how interconnect delay variations can be easily incorporated in the output deviations metric. We present the impact of interconnect delays and interconnect delay variations on test pattern selection, and the coverage of SDDs.

We start with a base set of \( n \)-detect transition delay-fault test patterns and apply our pattern-grading method to measure the effectiveness of each pattern. Next, we apply our pattern ordering method and select best patterns from the initial pattern set. Experimental results show that wire-delay variations must be taken into account in order to select high-quality patterns for today’s high performance designs. For
the same pattern count, the proposed pattern-grading and pattern-selection method
is more effective than a commercial timing-aware ATPG tool for SDDs, and requires
considerably less CPU time.

The remainder of the chapter is organized as follows. In Section 3.1, we show how
variations in wire delays can be incorporated in the output deviations framework.
In Section 3.2, we present experimental results for the IWLS 2005 benchmarks [43].
Section 3.3 summarizes this chapter.

3.1 Interconnect-Aware Output Deviations for Modeling Wire-Delay Variations

In this section, we show how to generalize the output deviations metric to include
wire-delay variations.
The wire delay variations and corresponding DDPs can be taken into consideration by assigning DDPMs to all wires. In this case, all wires can have a buffer-like DDPM. Various sources of wire-delay variations can be considered using the wire DDPM.

Consider the example in Fig. 3.1, which shows different metal layers and vias. Assume that M1 and M2 are local wires, and M3 and M4 are global wires. Assume that the net connecting port Q of CELL-1 to the input ports of CELL-2, CELL-3, and CELL-4 is called net $\xi$. We can model the wire delays in several different ways:

1. **Lumped-delay model for a net:** In this model, we can assign a single DDPM to each net. An example is given in Table 3.1. Assume that a signal with a signal-transition probability vector (STPV) of $\langle 0, 0.9, 0, 0 \rangle$ is propagated through net $\xi$ and does a $L \rightarrow H$ transition. Net $\xi$ has a DDP of 0.2 for the corresponding signal transition, so we update the STPV as follows:

$$STPV_{NEW} = 0.1 \odot \langle 1, 0, 0, 0 \rangle + 0.9 \odot \langle 0.2, 0.8, 0, 0 \rangle = \langle 0.28, 0.72, 0, 0 \rangle.$$

After updating the STPV, all the fanout gates use the updated probability values. This lumped model will have smaller impact on run-time compared to more detailed wire delay models. However, there is clearly a potential for incorrect output-deviation calculation. Consider the example in Fig. 3.1. CELL-1/Q to CELL-3/B has a single Metal-1 layer connection and expected to have a small delay variation in absolute terms. On the other hand, the path con-
Table 3.1: Example DDPM for net $\xi$ of Fig. 3.1

<table>
<thead>
<tr>
<th>prob</th>
<th>Initial Input State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0.2</td>
<td>0.3</td>
</tr>
</tbody>
</table>

Table 3.2: Example DDPM for each port-to-port path on net $\xi$ of Fig. 3.1

<table>
<thead>
<tr>
<th>prob</th>
<th>Initial Input State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>CELL-1/Q $\rightarrow$ CELL-2/A</td>
<td>0.08</td>
</tr>
<tr>
<td>CELL-1/Q $\rightarrow$ CELL-3/B</td>
<td>0.01</td>
</tr>
<tr>
<td>CELL-1/Q $\rightarrow$ CELL-4/A</td>
<td>0.16</td>
</tr>
</tbody>
</table>

necting CELL-1/Q to CELL-4/A includes global wires (M3, M4) and several vias. This path will clearly have a larger absolute delay variation compared to the path from CELL-1/Q to CELL-3/B. Thus, a more detailed delay model is required for higher accuracy.

2. Lumped delay model for a port-to-port path: In this model, we assign a single DDPM for each port to port path. An example is given in Table 3.2. Similar to the previous case, assume that a signal with a STPV of $\langle 0.1, 0.9, 0, 0 \rangle$ is propagated through net $\xi$ and does a $L \rightarrow H$ transition. STPVs for each path should be updated separately using the DDP values given in Table 3.2:

Table 3.3: Example DDPM for distributed delay model on net $\xi$ of Fig. 3.1

<table>
<thead>
<tr>
<th>(prob)</th>
<th>Initial Input State</th>
</tr>
</thead>
<tbody>
<tr>
<td>CELL-1/Q $\rightarrow$ $n_1$</td>
<td>0.01</td>
</tr>
<tr>
<td>$n_1$ $\rightarrow$ $n_2$</td>
<td>0.12</td>
</tr>
<tr>
<td>$n_1$ $\rightarrow$ CELL-4/A</td>
<td>0.15</td>
</tr>
<tr>
<td>$n_2$ $\rightarrow$ CELL-2/A</td>
<td>0.07</td>
</tr>
<tr>
<td>$n_2$ $\rightarrow$ CELL-3/B</td>
<td>0.01</td>
</tr>
</tbody>
</table>
As seen, the updated STPVs for the fanout ports have changed significantly for some of the ports.

3. **Distributed delay models:** In order to model the delay effects such as crosstalk on wires more accurately, we can use a distributed delay model and more than a single DDPM for each port to port path. For instance, in Fig. 3.1, assume that there is considerable delay variation between nodes $n_1$ and $n_2$ on M1, e.g., due to crosstalk. In this case, the path from CELL-1/Q to CELL-4/A will not be affected by the variation, but the other two paths will be impacted. We split the paths into smaller parts to account for this effect and create the DDPMs shown in Table 3.3. The new STPV values on fanout ports can be calculated as follows:

\[
STPV_{CELL-2/A} = 0.1 \odot \langle 1, 0, 0, 0 \rangle + 0.9 \odot \langle 0.08, 0.92, 0, 0 \rangle \\
= \langle 0.172, 0.828, 0, 0 \rangle.
\]

\[
STPV_{CELL-3/B} = 0.1 \odot \langle 1, 0, 0, 0 \rangle + 0.9 \odot \langle 0.01, 0.99, 0, 0 \rangle \\
= \langle 0.109, 0.891, 0, 0 \rangle.
\]

\[
STPV_{CELL-4/A} = 0.1 \odot \langle 1, 0, 0, 0 \rangle + 0.9 \odot \langle 0.16, 0.84, 0, 0 \rangle \\
= \langle 0.244, 0.756, 0, 0 \rangle.
\]
\[ STPV_{n_1} = 0.1 \odot (1, 0, 0, 0) + 0.9 \odot (0.01, 0.99, 0, 0) \]
\[ = (0.109, 0.891, 0, 0). \]
\[ STPV_{n_2} = 0.109 \odot (1, 0, 0, 0) + 0.891 \odot (0.12, 0.88, 0, 0) \]
\[ = (0.216, 0.784, 0, 0). \]
\[ STPV_{CELL-2/A} = 0.216 \odot (1, 0, 0, 0) + 0.784 \odot (0.07, 0.93, 0, 0) \]
\[ = (0.271, 0.729, 0, 0). \]
\[ STPV_{CELL-3/B} = 0.216 \odot (1, 0, 0, 0) + 0.784 \odot (0.01, 0.99, 0, 0) \]
\[ = (0.224, 0.776, 0, 0). \]
\[ STPV_{CELL-4/A} = 0.109 \odot (1, 0, 0, 0) + 0.891 \odot (0.15, 0.85, 0, 0) \]
\[ = (0.243, 0.757, 0, 0). \]

As expected, the delay variation between nodes \(n_1\) and \(n_2\) did not change the STPV of CELL-4/A appreciably, but it changed the STPV of other ports considerably. Distributed delay models can be made even more detailed by creating a new DDPM for each piece of metal and via. However, the more details added, the longer will be the run-time. As a result, the distributed delay model is useful for SDD detection only if it makes a considerable difference to the STPV values.

To incorporate crosstalk effects in the DDPM entries, we need to consider more details, since the delay induced by crosstalk depends on the direction and timing of signal-transitions on both the aggressor and victim wires. A DDPM lists the
DDPs that depend on signal transition of only the corresponding circuit element, e.g., gate or wire. In the case of crosstalk, a wire-DDPM depends to a large extent on a neighboring wire’s signal-transition characteristics. This problem can be solved by adding conditional DDP entries in wire-DDPMs for each victim wire. In this chapter, we focus on process variations on wires (the two lumped delay models) and leave the study of crosstalk effects (via a distributed delay model) to Chapter 4.

3.2 Experimental Results

In this section, we present experimental results obtained for the IWLS 2005 benchmark circuits. We do not consider the ISCAS benchmarks because these circuits are small and it is easier for an ATPG tool to excite all long paths with a small number of patterns. We first provide details of the experimental set-up. After that, we present the simulation results.

3.2.1 Experimental Set-up

All experiments were performed on a pool of state-of-the-art servers with at least eight processors available at all times, 16GB of memory, and running Linux. The program to compute output deviations was implemented using C++. Perl scripts were used to generate the simulation input files. A commercial tool was used to perform Verilog netlist synthesis and scan insertion for the IWLS benchmark circuits, which are available in Verilog RTL format [43]. Benchmark statistics are shown in Table 2.3. We used a commercial ATPG tool to generate $n$-detect TDF test patterns and timing-aware TDF patterns for these circuits. The ATPG tool was forced to
generate Launch-on-capture (LOC) transition fault patterns. The primary input change during capture cycles and the observation of primary outputs was prevented in order to simulate realistic test environments. Commercial tools were used for layout synthesis. Detailed wire-length data were extracted from the layout Design Exchange Format (DEF) files using an in-house Perl script. The path delays were calculated using an in-house dynamic path-timing simulator. All simulations were run in parallel on 8 processors.

3.2.2 Generating DDPMs for Gate Instances and Interconnects

DDPM of gate instances were generated by running 200 Monte Carlo (MC) simulations as described in Section 2.2.1. 180nm process technology parameters are used for HSpice simulations to match 180nm technology physical-design libraries. MC simulations were run using the following realistic process-variation parameters for a Gaussian distribution:

- Transistor gate length $L : 3\sigma = 10\%$
- Threshold voltage $V_{TH} : 3\sigma = 30\%$
- Gate-oxide thickness $t_{OX} : 3\sigma = 3\%$

For layout generation, we used a physical design library with 6 metal layers. Considering the library parameters, we assumed that M5 and M6 are global routing layers and all other metal layers are used for local routing. We used a delay variation of 30% on local wires and 10% on global wires, which increases with wire length.
Figure 3.2: The overall flow for layout-aware pattern selection.

as predicted in [34]. The effect of delay variations on vias is ignored and crosstalk effects will be studied in Chapter 4. The overall flow is shown in Fig. 3.2.2.

3.2.3 Results

In this subsection, we present the pattern-selection results using the number of excited distinct long paths as an evaluation metric.

An initial set of \( n \)-detect transition-fault patterns and a set of timing-aware transition-fault patterns were generated. For all values of \( n \), we implemented our method on the benchmarks to calculate output deviations. Then, we applied the pattern selection algorithm described in Section 2.1.4 by applying an output-deviation limit \((L)\) of 0.5. For each case, in order to perform a fair comparison, we set the
maximum number of patterns $S$ to be selected to the number of patterns in the
timing-aware transition-fault pattern set (max $S = \#$ timing-aware ATPG patterns).
Fig. 3.3 shows the normalized total CPU run-time for our method (sum of the CPU
times needed for $n$-detect pattern generation, deviation simulation, and pattern se-
lection) and the timing-aware ATPG. All CPU times are reported relative to the
run-time of timing-aware ATPG.

We find that even for large values of $n$, the total CPU time of the proposed
method is considerably less than the CPU time of timing-aware ATPG for the three
biggest benchmarks. For instance, the CPU time of aes_core benchmark is 503.8 s
for timing-aware ATPG , but only 142.13 s for the proposed method with $n=10$.
For tv80, there are a large number of hard-to-control paths and a large number
of fanouts. These factors reduces the efficiency of the $n$-detect pattern generation
procedure, hence there is an increase in the overall run-time of the proposed method.

Figs. 3.4-3.7 show the distribution of the run-time of the proposed method in
different steps, i.e., $n$-detect pattern generation, deviation simulation, and pattern
selection and reordering. As seen, for all values of $n$, $n$-detect pattern generation
takes longer time compared to deviation simulation, constituting the majority of the
CPU run-time. Pattern selection and reordering has negligible impact on the overall
run-time; the CPU time for this step is close to zero in all case, hence it is not shown
in Figs. 3.4-3.7. As $n$ increases, the ATPG step takes a larger portion of the total
CPU time.

We ran pattern quality evaluation analysis on a subset of complete benchmark set
due to resource limitations. Fig.3.8 shows the normalized number of excited distinct
long paths for selected patterns and the timing-aware ATPG patterns. We assume
Figure 3.3: The comparison of CPU run-time for the proposed method and timing-aware ATPG, using various values of $n$ on IWLS benchmarks.

that two paths are distinct if there is at least one non-shared gate instance. We also assume that any path with a nominal delay of larger than 70% of the clock period is a long path. We ran an in-house dynamic path-timing simulator on the benchmarks to determine the excited distinct long paths. All numbers are normalized by the number of long paths excited by timing-aware ATPG. Fig. 3.8 shows that the (unoptimized, academic implementation) output-deviation-based selection method finds patterns that excite a larger number of long paths compared to a commercial (and highly optimized) timing-aware ATPG tool, and with less CPU time. For the benchmark pci_bridge, the patterns selected from a 10-detect timing-unaware test set excite over 2.5x more long paths than timing-aware ATPG, where the proposed method excited 1408 long paths and the timing-aware ATPG excited only 536 long paths. For tv80, this ratio is 1.42 (921 vs. 650).

The proposed method achieves the excitation of more long paths with fewer test patterns in some cases. Table 3.4 shows the number of selected patterns for a subset
Figure 3.4: The distribution of the run-time over different steps of the proposed method, for $n=3$.

Figure 3.5: The distribution of the run-time over different steps of the proposed method, for $n=5$. 
Figure 3.6: The distribution of the run-time over different steps of the proposed method, for \( n=8 \).

Figure 3.7: The distribution of the run-time over different steps of the proposed method, for \( n=10 \).
of benchmarks. Note that the number of selected patterns may be less than the pattern count of timing-aware because we are dynamically dropping low-deviation patterns during deviation computation. Recall from Section 2.1.4 that we only retain the high-deviation patterns for reordering at the next step. When the pattern count is less than the expected value, we can run top-off ATPG to increase TDF fault coverage or to increase long path sensitization. Table 3.5 shows the number of top-off ATPG patterns that need to be added to the set of selected patterns. The required number of top-off patterns is very small compared to the number of selected patterns. Furthermore, for some cases, the proposed method does not need any top-off patterns, resulting in the same TDF coverage as timing-aware ATPG, but using less number of patterns (ac97_ctrl, aes_core, and pci_bridge).

Next we explain why timing-aware ATPG missed some long paths while the same ATPG tool when used without timing information excited more long paths. There are two main reasons for this apparent discrepancy. First, unspecified bits are randomly filled by the ATPG tool for both timing-aware and n-detect timing-unaware cases. Random fill may lead to the excitation of different paths. Second, in the timing-aware case, the ATPG tool attempts to activate a fault through a single long path, whereas n-detect ATPG tries to activate the same fault multiple times, through different paths. Unless it is forced with robustness options, the timing-aware ATPG tool does not re-try activating faults through longer paths. This implies that the ATPG tool dropped a fault as soon as it was activated through a long path, possibly not the longest path. If the robustness options are strictly used to force the tool to search for longest paths for all faults, the CPU-time of the timing-aware ATPG increases considerably.
Table 3.4: Number of selected patterns $S$

<table>
<thead>
<tr>
<th></th>
<th>$n=3$</th>
<th>$n=5$</th>
<th>$n=8$</th>
<th>$n=10$</th>
<th>timing-aware</th>
</tr>
</thead>
<tbody>
<tr>
<td>tv80</td>
<td>2021</td>
<td>2021</td>
<td>2021</td>
<td>2021</td>
<td>2021</td>
</tr>
<tr>
<td>ac97_ctrl</td>
<td>934</td>
<td>1284</td>
<td>1284</td>
<td>1284</td>
<td>1284</td>
</tr>
<tr>
<td>aes_core</td>
<td>2554</td>
<td>3657</td>
<td>5193</td>
<td>6156</td>
<td>7255</td>
</tr>
<tr>
<td>pci_bridge</td>
<td>2700</td>
<td>4163</td>
<td>4163</td>
<td>4163</td>
<td>4163</td>
</tr>
<tr>
<td>wb_conmax</td>
<td>4979</td>
<td>4979</td>
<td>4979</td>
<td>4979</td>
<td>4979</td>
</tr>
</tbody>
</table>

Table 3.5: Number of top-off patterns added to the selected patterns to achieve the same TDF coverage as timing-aware

<table>
<thead>
<tr>
<th></th>
<th>$n=3$</th>
<th>$n=5$</th>
<th>$n=8$</th>
<th>$n=10$</th>
</tr>
</thead>
<tbody>
<tr>
<td>tv80</td>
<td>169</td>
<td>263</td>
<td>263</td>
<td>263</td>
</tr>
<tr>
<td>ac97_ctrl</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>aes_core</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pci_bridge</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>wb_conmax</td>
<td>282</td>
<td>373</td>
<td>405</td>
<td>456</td>
</tr>
</tbody>
</table>

In order to further evaluate the relationship between output deviations and sensitized path lengths, we ran correlation analysis on the base $n$-detect patterns output deviations and their corresponding path delays. We used Matlab to compute Kendall’s correlation coefficients [62] for each pattern set. Similar to the results obtained in Section 2.2.5, Table 3.6 shows the average correlation coefficients for the patterns in a 10-detect test set of the IWLS’2005 benchmarks [43]. There is a strong positive correlation between output deviations and path lengths. It can be argued that a dynamic timing simulator can be used to obtain high correlation to path lengths. However, the method based on output deviations is flexible and general, and it can be used to account for many physical defects during pattern selection. Dynamic timing simulation can only provide variability-unaware timing information. The correlation between output deviations and path lengths is expected to be less
Figure 3.8: The normalized number of excited distinct long paths by selected patterns using output-deviation-based method (relative to timing-aware ATPG patterns).

Table 3.6: Kendall’s coefficients for evaluating the correlation of path lengths to output deviations

<table>
<thead>
<tr>
<th></th>
<th>Average</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>tv80</td>
<td>0.96</td>
<td>0.80</td>
<td>0.99</td>
</tr>
<tr>
<td>ac97_ctrl</td>
<td>0.90</td>
<td>0.73</td>
<td>0.96</td>
</tr>
<tr>
<td>aes_core</td>
<td>0.97</td>
<td>0.94</td>
<td>0.99</td>
</tr>
<tr>
<td>pci_bridge</td>
<td>0.90</td>
<td>0.88</td>
<td>0.99</td>
</tr>
<tr>
<td>wb_conmax</td>
<td>0.93</td>
<td>0.89</td>
<td>0.98</td>
</tr>
</tbody>
</table>

As more physical-defect data is integrated into deviation simulations. In this case, the method of output deviations is expected to reveal unique problematic paths that may be hidden from dynamic timing analysis.

As described in Section 1.2, the impact of local and global wires’ variations on circuit delay are different. In most designs, the local wires dominate global wires. We have determined the percentage of the local and global wires employed using our extraction tool for the above five benchmarks. Table 3.7 shows the distribution of local and global wires on long paths for each benchmark. It is evident that local routing layers dominate the wiring of long paths. Global wires are significant on
long paths only when the circuit size gets larger. For larger circuits, there are more global wires that are much longer than local wires, hence the impact of variations is expected to be higher.

We ran three different defect injection simulations to evaluate the fault-detection performance of the proposed pattern-selection method and the timing-aware ATPG. In the first defect injection simulation, in order to simulate process variation induced delay variations, we added Gaussian random delays on gates. We used a delay distribution with a standard deviation of 6% of the nominal delay of the corresponding gate. We obtained the approximate delay variation on gates from the MC simulations. In the second defect injection simulations, we also added Gaussian random delays on all nets. The additional delay on interconnects had a distribution with a standard deviation of 10% of the nominal delay of the corresponding interconnect. This delay distribution is selected using the results presented in [32,33]. Note that majority of the added delay defects is expected to be much less than the standard deviation values since the distribution is Gaussian. Thus, no single wire or gate delay variation is likely to create a visible delay defect on observation points. However, it is expected that these small delay variations may add up and cause visible delay defects on some of the paths. We used a clock period with a 3% slack from the

<table>
<thead>
<tr>
<th></th>
<th>Local Routing</th>
<th>Global Routing</th>
</tr>
</thead>
<tbody>
<tr>
<td>tv80</td>
<td>99.93%</td>
<td>0.07%</td>
</tr>
<tr>
<td>ac97_ctrl</td>
<td>100.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>aes_core</td>
<td>99.93%</td>
<td>0.07%</td>
</tr>
<tr>
<td>pci_bridge</td>
<td>100.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>wb_conmax</td>
<td>92.84%</td>
<td>7.16%</td>
</tr>
</tbody>
</table>
critical path.

For each benchmark, we created 1000 sample test cases, each with different random delay faults. We ran an abort-on-first-fail simulation on each sample using the selected patterns from various $n$-detect pattern sets and timing-aware ATPG patterns. Tables 3.8(a) and 3.8(b) show the number of failed samples for the first and second defect injection simulations using selected patterns and timing-aware ATPG patterns. For each entry, the results for the first case is followed by the results of the second case. As seen, the proposed method caught a considerably larger number of delay faults for most benchmarks in both cases. When wire delay variations are not considered, for aes_core benchmark, the additional delay was below the 3% clock period slack, therefore there are no detectable faults. However, injecting wire delay variations caused all the samples of aes_core to fail for all pattern sets. For tv80, the proposed method provides much better fault detection capability. Another key result obtained from the above defect injection simulations is that the proposed method has a much faster ramp-up in detecting faults: For tv80, the proposed method detected 849 (924) delay faults using the first 30 patterns, whereas timing-aware ATPG detected only 5 (8) delay faults after as many as 770 patterns.

We ran a third set of defect injection simulations to evaluate these methods for resistive open and resistive short defects (RORS) on interconnects. In order to simulate the larger delay impact caused by RORS, we injected a single delay defect on a randomly selected net. The injected defects have a normal delay distribution with a standard deviation of 15% of the clock period. Note that the injected faults are not guaranteed to cause a delay fault since they can hit a short path. We created 1000 distinct test cases for each benchmark. Table 3.8(c) shows the number of failed
Table 3.8: Number of delay faults detected using selected patterns and Timing-aware ATPG patterns (a) with delay variations only on gates (b) with delay variations on gates and interconnects (c) with RORS.

<table>
<thead>
<tr>
<th></th>
<th>n=3</th>
<th>n=5</th>
<th>n=8</th>
<th>n=10</th>
<th>timing-aware</th>
</tr>
</thead>
<tbody>
<tr>
<td>tv80</td>
<td>814</td>
<td>849</td>
<td>849</td>
<td>849</td>
<td>5</td>
</tr>
<tr>
<td>ac97_ctrl</td>
<td>450</td>
<td>450</td>
<td>450</td>
<td>450</td>
<td>450</td>
</tr>
<tr>
<td>aes_core</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>pci_bridge</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>wb_conmax</td>
<td>916</td>
<td>916</td>
<td>916</td>
<td>916</td>
<td>22</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>n=3</th>
<th>n=5</th>
<th>n=8</th>
<th>n=10</th>
<th>timing-aware</th>
</tr>
</thead>
<tbody>
<tr>
<td>tv80</td>
<td>878</td>
<td>922</td>
<td>924</td>
<td>924</td>
<td>8</td>
</tr>
<tr>
<td>ac97_ctrl</td>
<td>482</td>
<td>482</td>
<td>482</td>
<td>482</td>
<td>482</td>
</tr>
<tr>
<td>aes_core</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>pci_bridge</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>wb_conmax</td>
<td>968</td>
<td>968</td>
<td>972</td>
<td>972</td>
<td>22</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>n=3</th>
<th>n=5</th>
<th>n=8</th>
<th>n=10</th>
<th>timing-aware</th>
</tr>
</thead>
<tbody>
<tr>
<td>tv80</td>
<td>26</td>
<td>48</td>
<td>48</td>
<td>48</td>
<td>36</td>
</tr>
<tr>
<td>ac97_ctrl</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>aes_core</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>pci_bridge</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>wb_conmax</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
Figure 3.9: The number of detected delay faults for the proposed method and timing-aware ATPG for benchmark tv80.

samples using selected patterns and timing-aware ATPG patterns. As seen, the proposed method detected the same or more number of delay faults. Furthermore, we observed a faster ramp-up in delay fault detection for the proposed method. Fig. 3.9 shows how fast the delay faults are detected for tv80. As seen, the proposed method detects 24 delay faults within 1000 patterns, whereas timing-aware ATPG detects 18 delay faults after 2000 patterns.

Next, we selected patterns using the wire-delay-oblivious deviation model as presented in [51]. The objective here was to determine the extent of inaccuracy that results if interconnect delay variations are ignored. Table 3.9 shows the increase in the number of long paths excited for the lumped wire delay model (relative to the the wire-delay-oblivious model). We find that the addition of the wire delays has a considerable impact on the activation of long paths. For tv80, lumped-delay model excited over 30% more long paths compared to the wire-delay-oblivious model. The difference is around 20% for pci_bridge.
Table 3.9: The percentage increase in the number of long paths activated by the lumped-delay model relative to the wire-delay-oblivious model.

<table>
<thead>
<tr>
<th></th>
<th>n=3</th>
<th>n=5</th>
<th>n=8</th>
<th>n=10</th>
</tr>
</thead>
<tbody>
<tr>
<td>tv80</td>
<td>37.57%</td>
<td>38.70%</td>
<td>34.65%</td>
<td>27.74%</td>
</tr>
<tr>
<td>ac97_ctrl</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>aes_core</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>pci_bridge</td>
<td>15.09%</td>
<td>19.15%</td>
<td>20.26%</td>
<td>16.75%</td>
</tr>
<tr>
<td>wb_conmax</td>
<td>22.90%</td>
<td>4.28%</td>
<td>8.71%</td>
<td>-1.07%</td>
</tr>
</tbody>
</table>

We next ran the deviation computation and pattern selection steps using the pin-to-pin lumped delay model. Fig. 3.10 shows the total CPU run-time for the proposed method using the pin-to-pin delay model (sum of the CPU times needed for \( n \)-detect pattern generation, deviation simulation, and pattern selection), relative to the time needed for timing-aware ATPG. As seen, the overall run-time is still considerably less than the CPU time of timing-aware ATPG for the four biggest benchmarks. However, the need for additional computation increases the deviation calculation time. For aes_core and ac97_ctrl, the difference was negligible. However, more than 40% increase in run-time is observed for pci_bridge. This is mainly caused by the large number of fanouts on the internal nets of pci_bridge. Other benchmarks showed a 15-20% increase in run-time.

Next, we ran timing simulations on the selected patterns to determine the number of long paths excited using the pin-to-pin wire delay model. The results were very close to what we obtained for the lumped wire delay model, with a maximum difference of 3% between the two sets of results. This shows that, for the IWLS benchmarks, the additional computational complexity of the pin-to-pin wire delay model is not accompanied by any significant benefits in long-path activation. This might be because the fanout wires on nets are balanced in the layout, resulting in
very similar delays on all the fanouts of a net.

Finally, in order to evaluate impact of interconnect delay variations on new process technologies, we scaled the interconnect delays as foreseen by ITRS [32] to model a 45nm process technology. Table 3.10 shows the percentage of patterns that are selected for the 45nm technology, but not selected for the 180nm technology. The difference in these sets of patterns is quite small in all cases. However, when we examine the ordering of patterns, we find that the ordering of the patterns (based on deviations) are very different. These results imply that although the transition to 45nm technology did not change the long paths for a benchmark, it changed their relative importance considerably, hence criticality these paths.

Table 3.10: The percentage of different patterns selected at 180nm and 45nm process technologies.

<table>
<thead>
<tr>
<th></th>
<th>n=3</th>
<th>n=5</th>
<th>n=8</th>
<th>n=10</th>
</tr>
</thead>
<tbody>
<tr>
<td>tv80</td>
<td>2.23%</td>
<td>4.16%</td>
<td>4.16%</td>
<td>4.16%</td>
</tr>
<tr>
<td>ac97_ctrl</td>
<td>0.00%</td>
<td>1.80%</td>
<td>2.96%</td>
<td>3.19%</td>
</tr>
<tr>
<td>aes_core</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>pci_bridge</td>
<td>0.00%</td>
<td>0.12%</td>
<td>2.57%</td>
<td>4.44%</td>
</tr>
<tr>
<td>wb_conmax</td>
<td>1.59%</td>
<td>2.65%</td>
<td>3.86%</td>
<td>5.10%</td>
</tr>
</tbody>
</table>
3.3 Summary

We have presented a layout-aware pattern-selection technique for screening small-delay defects (SDDs) in nanometer integrated circuits. We have defined the concept of output deviations for interconnect-delay variations and pattern-pairs, and shown that it can be used as an efficient surrogate metric to model the effectiveness of transition delay-fault (TDF) patterns for SDDs. Experimental results for the IWLS 2005 benchmark circuits show that the proposed method selects an effective set of patterns for SDD detection from an \( n \)-detect TDF pattern set generated using a timing-unaware ATPG tool, and it excites a larger number of long paths compared to a commercial and highly-optimized timing-aware ATPG tool. The CPU time for the proposed method is also much less than that for timing-aware ATPG. The proposed method is computationally tractable, yet more accurate than a previous deviations-based method [51] that ignores chip layout and delay variations on interconnects.
Chapter 4

Consideration of Crosstalk

The most effective way of detecting an SDD is to detect it through long paths. The commercial ATPG tools and most previously proposed methods for detecting SDDs have relied on standard delay format (SDF) files generated during physical design flow [30, 40–42]. However, the path length is greatly impacted by process variations, crosstalk, and power supply noise. A path may became long because of such effects or a combination of them. Furthermore, the complexity of today’s ICs and shrinking process technologies have made design features more probabilistic. Thus, it is necessary to perform statistical timing analysis before evaluating the path length. We have covered process variation effects in earlier chapters. In this chapter, we consider crosstalk effects in addition to process variations during pattern evaluation and selection procedure.

The signal-transition probability method that was used in earlier chapters worked well for most cases. However, our analysis showed that in case of a large number of gates along the paths (deep combinational logic depth), output deviation metric can saturate and similar output deviations (close to 1) can be obtained for both long and intermediate paths (relative to clock cycle). In reality, the combinational logic depth of industrial circuits are usually shallow if the operating frequency is high. Nevertheless, an enhancement to the proposed probabilistic method is needed to overcome the problem of saturating output deviations.
In this chapter, we propose a statistical-timing-analysis based pattern grading and selection procedure to effectively target SDDs. The proposed method propagates a probability density function (pdf) along a sensitized path. Under the assumption of Gaussian delay distributions, the mean delay value of the path is therefore preserved without a saturation problem. Furthermore, the defect probability can also be extracted from the final Gaussian pdf as discussed in Section 4.1. We also updated our pattern selection method to increase the effectiveness of selected patterns using a smaller pattern count.

In our analysis, the impact of crosstalk, in addition to process variations, is taken into account dynamically during pattern evaluation. Such effects can change the length of a path and are capable of making a path longer, i.e. suitable for SDD detection. Similar to previous chapters, the objective is to select the best patterns for screening SDDs from a large repository test set, such as $n$-detect pattern set or randomly generated pattern set. The base pattern set, i.e., $n$-detect pattern set, is generated using traditional timing-unaware ATPG.

The remainder of this chapter is organized as follows. Section 4.1 presents the path delay analysis method that considers process variations and crosstalk. Pattern evaluation and selection procedure is presented in Section 4.2. In Section 4.3, we evaluate the proposed method. Section 4.4 concludes this chapter.

The work in this chapter is a collaborative research work that was jointly done by myself, Ke Peng, and our advisors Dr. Chakrabarty and Dr. Tehranipoor. The analysis experiments were performed by Ke Peng; Monte Carlo simulations and related data extractions, and coding of the sensitized path search tool was performed by me.
4.1 Delay Variations and Crosstalk Effects

In this section, we discuss the impact of process variations and crosstalk on circuit delay. We also present how these impacts are modeled in the proposed methodology.

4.1.1 Process Variations

In reality, the parameters of fabricated transistors are not exactly the same as design specification. In fact, the parameters are different from die-to-die, wafer-to-wafer, and lot-to-lot, and also between transistors on the same die. Parameter variations are caused by differences in impurity concentration densities, oxide thicknesses, diffusion depths, and similar factors. These nonuniform conditions result in deviations in transistor parameters, such as threshold voltage, W/L ratio, as well as variation in the widths of interconnect wires [63]. The process variations are expected to impact design performance (increase or decrease gate and interconnect delays) to a large extent in newer technologies [1]. Thus, it is necessary to take these effects into consideration in test pattern evaluation procedures.

Fig. 4.1 shows an example of our MC simulation results, using 250 MC simulations, on NAND3X1 gate in 180nm GSCLib3.0 technology library, under the following scenario:

1. Inputs 011 switch to 111,
2. Load capacitance equals to 100 pF,
3. Variation parameters ($3\sigma$):
   - $L = 27$ nm,
\( W = 27 \text{ nm}, \)

\( V_{TH0} = 0\%, 10\%, 15\%, \)

\( t_{OX} = 1\%, 2\%, 3\%. \)

As seen in Fig. 4.1, the delay variation is close to a Gaussian distribution. As the number of MC simulations increase, we expect the results to be much more closer to Gaussian distribution.

Based on MC simulation results, all process variations can be taken into account to calculate delay variations on each gate and interconnect. For simplicity, we assume that the delay variation of the segments on the path are independent from each other. Thus, independent Gaussian delay distributions are assumed for each circuit element (net or gate). Once the delay distributions of each gate and interconnect are found (presented as pdf in Fig. 4.2), we can calculate the pdf of the end point of a sensitized
Fig. 4.2 presents the pdf propagation through a target path. The pdfs of each path segment are shown below the path and the pdf of the complete path is shown above the path. Since independent Gaussian distributions are assumed, we can use Equations (4.1) and (4.2) to calculate the mean value and standard deviation of the path pdf [64,65].

\[
\mu_c = \sum_{i=1}^{N} \mu_i \quad (4.1)
\]

\[
\sigma_c = \sqrt{\sum_{i=1}^{N} \sigma_i^2} \quad (4.2)
\]

where \( \mu_i \) and \( \sigma_i \) are the pdf mean value and standard deviation of segment \( i \), respectively; \( \mu_c \) and \( \sigma_c \) are the path pdf mean value and standard deviation, respectively;
$N$ is the number of path segments.

Even if Gaussian distributions are not assumed for each delay segment, as long as segment delays are independent distributions with finite variances, the Central Limit Theorem (CLT) [64, 66] ensures that the path delay, which is the sum of segment delays, converges to a normal distribution [65, 67].

In case of reconvergent fanouts, the segment delays are not independent. For reconvergent fanouts, in order to evaluate the error introduced by Equation (4.2), we ran 20000 HSpice MC simulations on paths (with reconvergent fanouts) of different lengths. We observed 14% error in path pdf variance when the path includes only three gates. For six gates, the error was 12%, and for nine gates the error was found to be 9%. Since the proposed method targets longs paths, the expected error in long path variance calculation is approximately 10%. Developing a more accurate statistical method to cover reconvergent fanouts is left as a future research direction.

### 4.1.2 Crosstalk Effects

There are millions of interconnects running in parallel in the design, with parasitic coupling capacitance between them. Fig. 4.3 shows a small circuit in schematic representation. Fig. 4.4 shows the layout representation of the same circuit. As seen, may wires can be affecting each other even in small layout. Parasitic coupling impacts the circuit delay characteristics and performance. When there is a switching on both the target interconnect (victim net) and its neighboring net (aggressor net), there will be a crosstalk effect between them and the delay on both nets will be impacted [50, 68]. When the aggressor and victim nets have the same transition direction, delay on the victim net will be decreased (speed-up). Otherwise, when aggressor has
Figure 4.3: A small circuit used to highlight the crosstalk problem.

Figure 4.3: A small circuit used to highlight the crosstalk problem.

an opposite transition direction with respect to victim net, delay on the victim net will be increased (slow-down). Besides transition direction, transition arrival time between aggressor and victim nets also contributes to the crosstalk effects. If the aggressor and victim nets switch at the same time, the crosstalk effect is usually maximized. Otherwise, the crosstalk effect is reduced. Figs. 4.5 and 4.6 demonstrate crosstalk effects between two interconnects.

To understand the impact of signal arrival times on the crosstalk effect, we ran HSpice simulations. We modeled two nets with their parasitic parameters (coupling capacitance, load capacitance, resistance). These nets were not extracted from a design, but modeled as two generic nets in lower metal layers, e.g., M2. Next, 3000 HSpice simulations were run for each transition direction, and with different arrival time alignments between these nets. The arrival time difference was changed with a step size of 1ps. Each figure in Fig. 4.5 is drawn based on HSpice simulation results. Signal drive strengths were equal for each net. As a result, any of these nets can
Figure 4.4: Layout representation of the circuit in Fig. 4.3. Crosstalk effects are shown with arrows.
be considered a victim net, while the other one is aggressor. The $x$-axis in Fig. 4.5 represents arrival time difference between victim and aggressor nets, while the $y$-axis represents delay on the victim net. It is clearly seen that when aggressor and victim nets have the same transition direction (Fig. 4.5-a), the victim net will be sped up. The victim net will be slowed down when signals have opposite transition direction (Fig. 4.5-b). Furthermore, as seen in Fig. 4.5, the crosstalk effect on victim net delay is maximized when arrival time of aggressor and victim nets are approximately the same.

We ran HSpice simulations to understand the impact coupling capacitance on crosstalk effect. Similar to previous experiment, we modeled two generic nets with their parasitic parameters (coupling capacitance, load capacitance, resistance). Next, 2000 HSpice simulations were run with a different coupling capacitance between these nets. The coupling capacitance was changed with a step size of 0.05 fF. This experiment was repeated for various load capacitances to obtain Fig. 4.6. Fig. 4.6 is based on a fix arrival time difference (0) between these two nets. The $x$-axis in Fig. 4.6 represents the coupling capacitance between victim and aggressor nets, while the $y$-axis represents delay on the victim net. We find that the propagation delay increases linearly with the coupling capacitance between victim and aggressor nets.

In this chapter, we use curve-fitting method on simulation results to establish a functional relationship between interconnect delay and aggressor-victim alignment. We consider both transition direction and arrival time, as well as coupling capacitance between them. We update pdfs of interconnects along the sensitized paths using these results. We divided the crosstalk effect results obtained from previous HSpice simulations into different time windows as shown in Equation (4.3). Next,
Figure 4.5: Impact of aggressor arrival time on victim propagation delay when victim and aggressor nets have (a) same transition direction and (b) opposite transition direction.
Figure 4.6: Propagation delays when the coupling capacitance between victim and aggressor net increases assuming that both aggressor and victim nets have same arrival times.
for each region, we used Matlab curve-fitting toolbox to fit the simulation data to a polynomial function. A similar curve-fitting method was used to develop Equation (4.4), without dividing the data into timing windows.

The impact of arrival time on propagation delay is approximated in Equation (4.3).

\[
D_{\text{arrival}} = \begin{cases} 
  d, & 0 \leq t_{a-v} < t_1 \\
  a_0 t_{a-v} + a_1, & t_1 \leq t_{a-v} < t_2 \\
  b_0 t_{a-v} + b_1, & t_2 \leq t_{a-v} < t_3 \\
  d, & t_3 \leq t_{a-v} 
\end{cases} \tag{4.3}
\]

where \( D_{\text{arrival}} \) is the updated interconnect delay according to the impact of signal arrival time of its aggressor net, and \( d \) is the original interconnect delay without considering crosstalk effects. \( t_{a-v} \) is the arrival time difference between aggressor and victim nets. For the same transition direction between victim and aggressor nets, \( a_0 \) is negative and \( b_0 \) is positive. For opposite direction transition between them, \( a_0 \) would be positive and \( b_0 \) would be negative. The absolute values of \( a_i \) and \( b_i \) are different for different transition directions.

The impact of coupling capacitance on propagation delay is approximated in Equation (4.4).

\[
D_{\text{coup}} = a' d C_{a-v} \tag{4.4}
\]

where \( D_{\text{coup}} \) is the propagation delay considering the impact of coupling capacitance, \( d \) is the original interconnect delay without considering such effects, and \( C_{a-v} \) is the coupling capacitance between aggressor and victim nets. The factor \( a' \) is negative.
Figure 4.7: Crosstalk impact on interconnect *pdfs*.

for same transition direction case and positive for opposite transition direction case.

Figure 4.7 shows the crosstalk impacts on interconnect *pdfs*. In our work, we assume that for each interconnect, crosstalk effects will only impact its mean delay value, rather than delay variation. In this way, only the mean delay value of the interconnect will be updated as a result of crosstalk effects, as showing in Fig. 4.7.

### 4.2 Pattern Evaluation and Selection Method

From our discussion in Section 4.1.1, in general, a TDF pattern would be considered more efficient in detecting SDDs when it sensitizes a large number of long paths. Thus, it is necessary to identify all the paths sensitized by each pattern to evaluate its effectiveness. We developed an in-house tool to list all the sensitized paths for a TDF pattern in addition to each segment of the sensitized paths. This tool enumerates all paths sensitized by a given test pattern. The tool works as follows:

- Commercial ATPG tools runs ATPG and fault simulation steps.
- For each pattern, the ATPG tool reports the detected transition-delay faults.

This report includes the net name, as well as the signal transition type, i.e.,
falling or rising.

- Our in-house tool finds the active nets inside the CUT. This step has $O(\log N)$ complexity.

- Starting from scan flip-flops and primary inputs, each net with a detected fault is traced forward. Note that if a fault is detected, it means that this net reaches a scan flip-flop through other nets with detected faults. If the sensitized path has no branching on it, the complexity of this step is $O(N)$. However, if there are $K$ branches on the sensitized net, and if all these branches create a different sensitized path, the complexity of this step is $O(N^K)$.

- Note that, although unlikely, if a test pattern can test all nets at the same time, the run time of sensitized path search procedure is very high. However, our simulations on academic benchmarks and industrial circuits showed that, for most cases, a maximum of 5-10% of the nets can be tested for transition-fault by a single test pattern, and the sensitized paths have a very small number of branching. As a result, the run time of sensitized path search procedure is considerably less than the ATPG run time. The run-time results are presented in Section 4.3.

The tool report includes all sensitized paths and their segments. For example, $Pattern_i$ sensitizes $M_i$ paths $Path_{i1}, Path_{i2}, \cdots, Path_{iM_i}$, and for each of its path $Path_{ij}$ ($j \in [1, M_i]$), we list the path segments $Seg_{ij1}, Seg_{ij2}, \cdots, Seg_{ijl_{ij}}$. We set up a database of pdfs for each gate and interconnect based on MC simulations. Next, the interconnect delay is updated considering crosstalk effects as discussed in
Section 4.1.2. The pdf of each path is calculated using Equations (4.1) and (4.2).

In Section 4.2.1, we discuss path analysis and path evaluation after getting the path pdf. After calculating the weight of each sensitized path of pattern\(_i\), the weight of pattern\(_i\) (\(W_{\text{pattern}_i}\)) is calculated by summing all the weights using Equation (4.5).

\[
W_{\text{pattern}_i} = \sum_{i=1}^{M_i} W_{\text{path}_i}
\]  

where \(M_i\) is the total number of sensitized paths by pattern\(_i\).

Finally, after pattern evaluation, we will discuss pattern selection procedure in Section 4.2.2.

### 4.2.1 Path pdf Analysis

As mentioned earlier, Gaussian distribution is assumed for each gate and interconnect delay and MC simulations are used to obtain the pdfs of each path segment (gate or interconnect). Interconnect delay is updated considering crosstalk effects. Using the delay data, the pdf of the sensitized path can be calculated. Each path is evaluated by its comparable length to clock cycle. During pattern analysis, we first obtain the clock cycle \([0, T]\), and the timing windows based on the clock cycle. Fig. 4.8 shows an example of timing windows for two different paths, \(i\) and \(j\). In Fig. 4.8, there are four timing windows for path \(i\): \(W_i1 = [0, \Delta_1 T]\), \(W_i2 = [\Delta_1 T, \Delta_2 T]\), \(W_i3 = [\Delta_2 T, T]\), and \(W_i4 = [T, +\infty]\). Using these timing windows, delay defect probabilities \(P_i1, P_i2, P_i3\) and \(P_i4\) for \(W_i1, W_i2, W_i3\) and \(W_i4\) can be calculated, respectively. Then, the weight of path \(i\) (\(W_{\text{path}_i}\)) is calculated using Equation (4.6).
where $\alpha_1$, $\alpha_2$, $\alpha_3$, and $\alpha_4$ are weights assigned to defect probability regions $P_i1, P_i2, P_i3$ and $P_i4$, respectively.

$W_{pathi}$ represents the probability that the path is longer than clock cycle if $\alpha_1$, $\alpha_2$, and $\alpha_3$ are set to 0 and $\alpha_4$ is set to 1. If $\alpha_1$ and $\alpha_2$ are set to 0, $\alpha_3$ and $\alpha_4$ are set to 1, $W_{pathi}$ represents the probability that the path is longer than $\Delta_2T$. Therefore, Equation (4.6) is very flexible in calculating path weight.

Fig. 4.9 shows an example of path weight and pattern weight calculation. In this example, $Pattern_j$ sensitizes four different paths. pdfs of each path is shown in Fig. 4.9 as $PDF1$, $PDF2$, $PDF3$, and $PDF4$, respectively. If we consider long
paths as longer than 70% of the clock cycle, we can set $0.7T$ as our long path threshold. In this case, we set $\alpha_1 = \alpha_2 = 0$, $\alpha_3 = \alpha_4 = 1$, $\Delta_2 = 0.7$. $\Delta_1$ can omitted since $\alpha_1 = \alpha_2 = 0$. In this way, $0.7T$ becomes the path calculation threshold as well. The weight of a path is defined as the probability that the path is longer than this threshold. The weights of these four paths can be calculated as $W_{\text{path}1} = 1$, $W_{\text{path}2} = 0.65$, $W_{\text{path}3} = 0.3$, and $W_{\text{path}4} = 0$. Then the weight of $\text{Pattern}_j$ can be calculated using Equation (4.5): $W_{\text{pattern},j} = 1 + 0.65 + 0.3 + 0 = 1.95$. Since we use pdfs of each sensitized paths, rather than a fixed delay value, the long path should be defined in a slightly different way. If the weight of a path is larger than 0.5, i.e., if the mean value of the path pdf is larger than this threshold, it is considered to be a long path.

**Figure 4.9**: An example of path evaluation and pattern evaluation.
4.2.2 Pattern Selection

From our analysis and pattern weight definition in the previous sections, if a pattern has a large weight, it is more efficient in detecting SDDs. Our analysis has shown that there is a high correlation between pattern weight and the number of sensitized long paths. Thus, we select the patterns with largest weights. However, it is possible that, some of the sensitized paths of two different patterns are the same. If a path has already been detected by the selected patterns, it is not necessary to use it for evaluating the remaining patterns. The objective of this method is to only include the number of unique long paths sensitized by each pattern during pattern weight calculation.

In the proposed pattern selection procedure, the largest weighted patterns is ordered to be the first to be selected. After selecting the pattern, we re-calculate the weight of all the remaining patterns by excluding paths detected by the selected pattern. Then, the pattern with largest weight in the remaining pattern set is selected. This procedure is repeated until some stopping criteria is met, for instance the largest pattern weight is smaller than a specific threshold. This pattern selection procedure will ensure that the best patterns can be selected from an n-detect pattern set. It can also ensure that there are as little overlap as possible between patterns in terms of sensitized paths. Therefore, it can further reduce the pattern count.

The pattern-sorting algorithm is shown in Fig. 4.10. In this algorithm, each pattern is evaluated by not-detected paths, i.e., that are not sensitized by the previously selected patterns. This algorithm will return a pattern list with decreasing order according to the test efficiency of patterns, using which we can set a threshold and
01: \( S \leftarrow \text{pattern list of all the patterns} \)
02: \( S = \{ P_1, P_2, \ldots, P_N \} \)
03: \( W(P_i) \leftarrow \text{weight of pattern } i, P_i \)
04: \( P_s = \text{NULL} \)
05: \( \text{for } (i = 1, \ldots, N) \)
06: \( \{ \)
07: \( \quad \text{for } (j = i, \ldots, N) \)
08: \( \quad \{ \)
09: \( \quad \quad \text{update } W(P_j) \text{ by removing paths detected by } P_s \)
10: \( \quad \} \)
11: \( P_{\text{max}} = \text{MAX}\{P_j, \ldots, P_N\} \)
12: \( \text{if } (W(P_i) < W(P_{\text{max}})) \)
13: \( \quad \{ \)
14: \( \quad \quad \text{exchange } P_i \text{ and } P_{\text{max}} \text{ in pattern list } S \)
15: \( \quad \} \)
16: \( P_s = P_i \)
17: \( \} \)
18: \( \text{Return sorted pattern list for pattern selection.} \)

Figure 4.10: Pattern sorting algorithm.

select the best patterns from the \( n \)-detect pattern set.

4.3 Experimental Results

In this section, we present experimental results obtained for some of the ISCAS and IWLS benchmark circuits. We first provide an overview of the pattern evaluation and selection flow (Section 4.3.1). Next, we provide details for the benchmark circuits and the experimental setup (Section 4.3.2). Last, we present the simulation results (Sections 4.3.3 and 4.3.4).
4.3.1 Pattern Evaluation and Selection Flow

The complete pattern evaluation and selection flow is shown in Fig. 4.11. In our experiments, Design Compiler was used for circuit synthesis and Astro was used to perform the placement and routing of standard cells. After physical synthesis on the design, $n$-detect ATPG is run to generate source patterns for our pattern evaluation and selection procedure. Parasitic parameters of the design was extracted from the layout for crosstalk analysis. As we discussed in the previous section, crosstalk analysis is used to update the interconnect delay pdfs in the sensitized paths. Next, the original pdfs of each gate and interconnect, considering process variations, are obtained from SPICE simulation. Then, with the path calculation threshold definition, we perform the pattern evaluation and pattern selection steps to select the most efficient patterns from the $n$-detect pattern set. Finally, top-off ATPG is run to detect the undetected faults in the design and meet the fault coverage.
requirement.

4.3.2 Experimental Setup

Physical synthesis was run using GSCLib180 Standard Cell Library. PrimeTime SI was used for crosstalk analysis. A custom in-house tool is developed to report the sensitized paths for each pattern.

We performed our experiments on several Linux x86 servers with 8 processors and 24 GB of available memory. Commercial tool PrimeTime SI was used to extract the victim-aggressor pairs from the design layout. The programs for performing crosstalk calculation, pattern evaluation and selection were implemented using C. TetraMAX was used to generate 1-detect, n-detect, and timing-aware TDF patterns for our benchmark circuits. All generated patterns are Launch-of-Capture (LOC) transition fault patterns. 7 IWLS benchmarks and 2 ISCAS benchmarks were involved in our experiments. The details of these benchmarks are shown in Table 4.1. Number of logic gates is shown in the second column, and number of flip-flops is shown in the third column. The total number of standard cells is presented in the fourth column. Note that the data in Table 4.1 are may slight change while synthesizing the circuits, since the tool may optimize the design according to user’s optimization options.

4.3.3 Pattern Selection Efficiency

In this subsection, we present experimental results for validating the pattern selection procedure. We validate our pattern selection procedure by counting the total number of sensitized unique long paths of the selection patterns. Our validation procedure depends on the assumption that if a pattern can detect many long paths, it will
Table 4.1: Details of Experimental Benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># of gates</th>
<th># of FFs</th>
<th># of Total standard cells</th>
</tr>
</thead>
<tbody>
<tr>
<td>tv80</td>
<td>6802</td>
<td>359</td>
<td>7,161</td>
</tr>
<tr>
<td>ac97_ctrl</td>
<td>9,656</td>
<td>2,199</td>
<td>11,855</td>
</tr>
<tr>
<td>mem_ctrl</td>
<td>10,357</td>
<td>1,083</td>
<td>11,440</td>
</tr>
<tr>
<td>systemcaes</td>
<td>7,289</td>
<td>670</td>
<td>7,959</td>
</tr>
<tr>
<td>wb_dma</td>
<td>2,826</td>
<td>563</td>
<td>3,389</td>
</tr>
<tr>
<td>s13207</td>
<td>886</td>
<td>333</td>
<td>1,219</td>
</tr>
<tr>
<td>s9234</td>
<td>829</td>
<td>145</td>
<td>974</td>
</tr>
</tbody>
</table>

also detect many SDDs. Experimental results show that there is a high correlation between the number of sensitized long paths and the number of detected SDDs.

Similar to the example in Section 4.2.1, we set \( \alpha_1 = \alpha_2 = 0 \), \( \alpha_3 = \alpha_4 = 1 \), and \( \Delta_2 = 0.7 \), so that the path calculating threshold is 0.7x clock cycle. We consider a path is a long path if its weight is larger than 0.5, i.e., the mean value is larger than this long path threshold.

Fig. 4.13 shows the long path coverage ramp up for tv80 benchmark. For tv80 benchmark, the pattern count of \( n \)-detect pattern set (\( n = 10 \)) is 11,072, and the number of sensitized long paths is 1,718. As seen in Fig. 4.13, only 520 patterns (4.7% of the original pattern count) can detect all the long paths that is sensitized by the \( n \)-detect pattern set. Similarly, 315 patterns (2.8% of the total pattern count) can detect 1,516 long paths (90.0% of all sensitized long paths). From this experimental result, we can see that the proposed pattern grading and selection procedure can be very efficient in selecting patterns.
4.3.4 Pattern Set Comparison

In our pattern selection procedure, a pattern weight threshold is set to stop the procedure. For example, the pattern weight threshold in this subsection is 1, i.e., only the patterns with weight larger than 1 can be selected. Then top-off ATPG is run to detect the remaining faults and meet the test coverage requirement. As a result, the final pattern set of our procedure is the selected pattern set plus top-off ATPG pattern set.

From our discussion in previous sections, the $n$-detect pattern set and timing-aware pattern set will do a better job in sensitizing long paths and detecting SDDs. Experimental results in Table 4.2 and Table 4.3 demonstrate the correctness of this conclusion. Columns 2, 3, 4 and 5 in Table 4.2 present the number of sensitized long paths and number of detected SDDs for tv80.

Figure 4.12: Correlation between the number of sensitized long paths and number of detected SDDs for tv80.
unique long paths of 1-detect, 5-detect, 10-detect, and timing-aware pattern set, respectively. Column 6 presents the number of sensitized long paths of our selected pattern set. Column 7 presents the number of long paths sensitized by top-off ATPG pattern set but not sensitized by the selected pattern set. Column 8 presents the total number of sensitized long paths of our final pattern set, i.e., the selected patterns plus top-off ATPG patterns. From the results in Table 4.2, we can see that timing-aware ATPG and $n$-detect ATPG pattern sets are always detecting significantly more number of long paths than 1-detect pattern set, except S13207 benchmark. On the other hand, timing-aware ATPG is not as effective as 10-detect ATPG in long path sensitization. However, our pattern set, i.e., selected patterns plus top-off ATPG
patterns, is more effective than 10-detect ATPG pattern set in terms of long paths sensitization, except S9234 benchmark. The number of sensitized long paths are very close for S9234.

Table 4.3 presents the number of SDDs for different pattern sets. SDDs are defined as TDFs on the long paths. The results are similar to long paths sensitization result. This is expected since SDDs are TDF on the long paths. If a pattern detects many long paths, it can also detects many SDDs. As seen in Fig. 4.12, there is a very high correlation between the number of sensitized long paths and number of detected SDDs.
Table 4.4: Number of patterns for different pattern set.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>n=1</th>
<th>n=10</th>
<th>t.aware</th>
<th>selected</th>
<th>top-off</th>
<th>sel+top-off</th>
</tr>
</thead>
<tbody>
<tr>
<td>tv80</td>
<td>1,435</td>
<td>11,072</td>
<td>17,107</td>
<td>393</td>
<td>924</td>
<td>1,317</td>
</tr>
<tr>
<td>ac97 ctrl</td>
<td>1,032</td>
<td>6,393</td>
<td>4,087</td>
<td>126</td>
<td>834</td>
<td>960</td>
</tr>
<tr>
<td>mem ctrl</td>
<td>1,595</td>
<td>13,142</td>
<td>6,577</td>
<td>352</td>
<td>1,032</td>
<td>1,384</td>
</tr>
<tr>
<td>systemcaes</td>
<td>591</td>
<td>3686</td>
<td>5,590</td>
<td>800</td>
<td>30</td>
<td>830</td>
</tr>
<tr>
<td>wb_dma</td>
<td>483</td>
<td>3,600</td>
<td>4,460</td>
<td>131</td>
<td>354</td>
<td>485</td>
</tr>
<tr>
<td>s13207</td>
<td>810</td>
<td>6,712</td>
<td>1,108</td>
<td>36</td>
<td>775</td>
<td>811</td>
</tr>
<tr>
<td>s9234</td>
<td>343</td>
<td>2,763</td>
<td>428</td>
<td>64</td>
<td>271</td>
<td>335</td>
</tr>
</tbody>
</table>

Table 4.4 presents the number of patterns of 1-detect, 5-detect, 10-detect, timing-aware ATPG, and our pattern set. These patterns are used for the long path and SDD detection experiment that are summarized in Table 4.2 and Table 4.3. From these results, we can see that timing-aware ATPG will result in a large number of patterns compared to 1-detect pattern set for large IWLS benchmarks. For some cases, its pattern count is even larger than 10-detect pattern set, e.g., tv80 and wb_dma benchmarks. For all cases, our pattern set, i.e., selected patterns plus top-off patterns, would result in a significantly small number of patterns compared with n-detect pattern set or timing-aware pattern set. In short, our pattern set can detect a large number of long paths with very much close to 1-detect pattern count.

### 4.3.5 Long Path Threshold Analysis

Long path threshold is an important parameter for our procedure. If the long path threshold changes, the path weight calculation threshold will be changed accordingly, and further the weight of each pattern will be re-evaluated. Although it will not change the comparable importance of the patterns, it will impact the number of
Table 4.5: Long path threshold impact on pattern selection results for tv80 benchmark.

<table>
<thead>
<tr>
<th>$L_P^{thr}$</th>
<th># of sel patterns</th>
<th># of top-off patterns</th>
<th>Total # patterns</th>
<th># of LPs</th>
<th># of SDDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.7T</td>
<td>393</td>
<td>924</td>
<td>1,317</td>
<td>2,236</td>
<td>82,502</td>
</tr>
<tr>
<td>0.8T</td>
<td>60</td>
<td>1,279</td>
<td>1,765</td>
<td>63,428</td>
<td></td>
</tr>
</tbody>
</table>

selected patterns in the pattern selection procedure, number of detected long paths and SDDs, according to our definitions above. If the long path threshold increases, the number of selected patterns decreases, and the number of top-off ATPG patterns increases to meet the fault coverage requirement. On the other hand, if we decrease the long path threshold, the number of selected patterns increases and top-off ATPG pattern count decreases. The number of sensitized long paths and SDDs will also changed accordingly.

The results in Section 4.3.4 is only for fixed long path threshold (0.7x clock cycle). Table 4.5 presents experimental results with two different long path thresholds (0.7T to 0.8T) for the tv80 benchmark. When the long path threshold increased, the weight of each pattern will be reduced, and the number of selected pattern will be decreased. In this case, more top-off patterns would be generated to meet the fault coverage requirement. These are shown in Columns 2 and 3 in Table 4.5. The number of sensitized long paths and detected SDDs will be decreased due to the increase in long path threshold $L_P^{thr}$, as shown in Columns 5 and 6.
4.4 Summary

We have presented a process variations and crosstalk-aware pattern-selection technique for screening small-delay defects (SDDs). We presented an improved probability propagation scheme that solves the problem of output-deviations saturation. A new pattern selection method is also presented, which is shown to be very efficient in selecting the most effective patterns from a large repository of test patterns. Experimental results for the IWLS 2005 and ISCAS 89 benchmark circuits show that the proposed method clearly outperforms commercial timing-aware ATPG in both pattern quality and the number of test patterns. Using an order-of-magnitude fewer test patterns, the proposed method sensitizes more long paths than timing-aware ATPG.
Chapter 5

Seed Selection in Test Compression

The complexity of today’s integrated circuits and shrinking process technologies are leading to prohibitively high test data volumes. As a result, the 2007 ITRS document predicts that the test data volume for integrated circuits will be as much as 38 times larger and the test application time will be about 17 times larger in 2015 than today [1]. It has also been shown recently that for sequence- and timing-dependent faults leads to high test-data volume; for example, the test-data volume for delay faults is 2-5 times higher than that for stuck-at faults [44]. Test-data compression is therefore essential to reduce test-data volume and testing time.

Test compressions techniques have been extensively studied to reduce test-data volume [69–73]. Many of these compression techniques rely on the use of a linear-feedback shift register (LFSR) or an XOR network [69,70]. These methods rely on the fact that test cubes contain very few specified bits. However, these techniques are typically tailored towards the embedding of test cubes for stuck-at faults, and they do not target sequence- and timing-dependent faults. Typically, unspecified bits in a generated seed are randomly filled without further evaluation of the quality of generated test patterns for the detection of timing-related defects. Therefore, new test-data compression methods are required to reduce test-data volume while targeting SDDs.

LFSR reseeding has long been recognized as an efficient test compression tech-
A test cube can be encoded with high probability into a compact seed of typical length $S_{\text{max}} + 20$, where $S_{\text{max}}$ equals the number of specified bits in the test cube [74]. A seed can be computed for each test cube by solving a system of linear equations based on the feedback polynomial of the LFSR. The solution space for the system of linear equations is quite large, especially for test cubes with few specified bits. All patterns derived from the possible solutions (LFSR seeds) cover the original test cube. However, these patterns differ from each other in their ability to detect SDDs. Therefore, it is important to select LFSR seeds that can be used to target a large number of defects. Moreover, if more effective seeds are loaded first into the LFSR, a steeper coverage ramp-up can be obtained.

The choice of LFSR seeds for compression methods is either random, e.g., as an outcome of Gauss-Jordan Elimination, or seed selection is designed for better seed compression [71]. These methods do not target the coverage of SDDs. To enhance the effectiveness of LFSR reseeding for process variations and the resulting SDDs, new techniques are needed for pattern modeling and seed selection. While recent work has also focused on seed selection for increasing defect coverage [48], it is not applicable to SDDs since it only evaluates the effectiveness of single patterns, instead of pattern-pairs and long-path sensitization that are required for SDD detection.

In this chapter, we use an LFSR-reseeding-based test compression technique to detect SDDs by selecting the best LFSR seeds, and by ensuring that the resulting patterns sensitize more long paths in the circuit. The proposed method is based on the use of the output-deviations metric for delay faults, which was introduced in Chapter 2. As in 2, the output deviation measure is used as a surrogate coverage-metric for SDDs. The seeds are selected to ensure high output deviations and high
Procedure: Find Seeds $R(T,N_s,N_p)$

1: list $R[|T|]$;
2: for all test cube $t_p$, $p = 1, 2, ..., |T|$ do
3: list $S[N_s], V[N_s]$;
4: for $i = 1$ to $N_s$ do
5: generate a new seed $s_{p,i}$;
6: expand seed $s_{p,i}$ into test pattern $v_{p,i}$;
7: add $s_{p,i}$ to $S$, add $v_{p,i}$ to $V$;
8: end for
9: $D_p = \text{Compute} \_\text{Deviations}(S, V, N_p)$;
10: $R_p = max(D_p)$; add $S[R_p]$ to $R$;
11: end for
12: return $R$;

Figure 5.1: Procedure to select LFSR seeds.

coverage of least-slab (long) paths. Simulation results for the IWLS 2005 benchmark circuits [43] show that compared to several baseline seed-selection methods, the proposed method leads to patterns that provide higher coverage of long paths. In addition, the selected patterns also provide steeper ramp-up of long-path coverage as well as higher coverage of injected delay-defects.

The remainder of the chapter is organized as follows. Section 5.1 describes the proposed seed-selection procedure. In Section 5.2, we present experimental results for the IWLS 2005 benchmarks. Section 5.3 summarizes this chapter.

5.1 Seed Selection

In this section, we describe how to use output deviations to select high-quality seeds for the LFSR. The solution of the system of linear equations in any LFSR reseeding scheme typically leads to partially-specified seeds, hence there are many choices for selecting a seed for a given test cube.
A high-level description of the LFSR seed-selection algorithm is shown in Fig. 5.1. Given a set of test cubes $T$, we generate $N_s$ seeds for each test cube. Then, each test cube $t_p$ is expanded into a fully-specified test pattern $t_i$. The seeds and their corresponding patterns are stored in lists $S$ and $V$, respectively (lines 5-10). In the next stage, deviation computation is done for the patterns in $V$ (line 11).

We use the deviation-computation algorithm described in Section 2.1.4 using a deviation-limit of $P = 0.5$ and an effective-list size of 5. In the next step, we select the most effective seed and move to the next test cube. Once all the seeds are specified, the final fully-specified patterns are also sorted on the basis of the pattern effectiveness calculated during the deviation computation. In this way, the patterns that cover wider range of long paths are pushed to the top of the pattern list. We expect the eventual fault-coverage analysis to reveal that many lower-ranked patterns will not be needed and the patterns at the end of the list can be discarded after all the don’t-care bits in the seeds are specified.

**Example 1:** Fig. 5.2 shows an external-XOR LFSR with feedback polynomial $x^5 + x^3 + x + 1$, and a 1-stage phase shifter. The state of the LFSR can be represented using a vector $\vec{S} = (s_1, s_2, \ldots, s_N)^t$, where $N$ is the size of the LFSR and $s_1(s_N)$ corresponds to the leftmost (rightmost) stage. The $j^{th}$ state of the LFSR is derived recursively as $\vec{S}_j = H \vec{S}_{j-1}, j = 1, 2, \ldots$, where $H$ is the state transition matrix for the LFSR. The $j^{th}$ output of the 1-stage phase shifter shown in Fig. 5.2 can be represented as $O_j = \vec{P}^t \vec{S}_j = \vec{P}^t H^j \vec{S}_0, j = 1, 2, \ldots$. Vector $\vec{P}$ represents the operation of the phase shifter. If stage $j$ of the LFSR is connected to the XOR gate, the $j^{th}$ row in $\vec{P}$ is set to ‘1’. For the phase shifter in Fig. 5.2, we have $\vec{P} = (10100)^t$. 

165
For example, the second output of the phase shifter is $O_1 = \vec{P}^t H S_0 = y_1 + y_2 + y_3 + y_5$.

For the test cube 101xxxxx (the leftmost bit ‘1’ is loaded into the first scan cell that is next to the scan out pin), we can obtain a system of linear equations as shown in Fig. 5.3(a). Gauss-Jordan Elimination can be used to transform a set of columns in A into an identity matrix (these columns are referred to as pivots) while the remaining columns are free-variables, as shown in Fig. 5.3(b). The set of solutions for the pivots can be represented as a linear combination of the free-variables, as shown in Fig. 5.3(c). To obtain multiple seeds for each test cube, we first make random assignments to free-variables and then compute pivots.

If we assume that $N_s = 2$, we can randomly assign bit values to $y_4$ and $y_5$ to find 2 different seeds: If $y_4 = 1$ and $y_5 = 1$, we find $(y_1, y_2, y_3) = (0, 0, 1)$. Similarly, if $y_4 = 0$ and $y_5 = 0$, we find $(y_1, y_2, y_3) = (1, 1, 0)$. Once we find the complete set of seeds, we can start the deviation computation for the expanded patterns. Assume that we set $N_p = 1$ and the circuit under test has 20 observation points. If seed-1 creates largest deviation for 15 observation points, and seed-2 creates largest deviation for 5 observation points, we select seed-1, i.e., $(y_1, y_2, y_3, y_4, y_5) = (0, 0, 1, 1, 1)$. Note that $N_p$ is smaller than the number of candidate seeds.

5.2 Experimental Results

In this section, we present experimental results obtained for the IWLS 2005 benchmark circuits. We do not consider the ISCAS benchmarks because these circuits are small and it is easier for an ATPG tool to excite all long paths with a small number of patterns. We first provide details of the experimental set-up. After that,
\[ H = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \]

**Figure 5.2:** (a) An LFSR and phase shifter; (b) State transition matrix of the LFSR.

\[
\begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}
\]

(a) System of linear equations

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

(b) Gauss-Jordan elimination

(c) Solution space

**Figure 5.3:** Example to illustrate the solution for system of linear equations.

we present the simulation results.

### 5.2.1 Experimental Set-up

All experiments were performed on a pool of state-of-the-art servers running Linux. The program to generate seeds, compute output deviations, and sort patterns was implemented using C++. A commercial tool was used to perform Verilog netlist synthesis and scan insertion for the IWLS benchmark circuits, which are available in Verilog RTL format [43]. Benchmark statistics are shown in Table 5.1 (synthesized
Table 5.1: Benchmark statistics

<table>
<thead>
<tr>
<th></th>
<th># I/O</th>
<th># flip-flops</th>
<th># library cells</th>
<th># specified bits (max)</th>
</tr>
</thead>
<tbody>
<tr>
<td>wb_dma</td>
<td>1969</td>
<td>881</td>
<td>6783</td>
<td>52</td>
</tr>
<tr>
<td>tv80</td>
<td>763</td>
<td>359</td>
<td>7435</td>
<td>120</td>
</tr>
<tr>
<td>mem_ctrl</td>
<td>2537</td>
<td>1138</td>
<td>11119</td>
<td>129</td>
</tr>
<tr>
<td>usb_funct</td>
<td>3740</td>
<td>1766</td>
<td>17495</td>
<td>83</td>
</tr>
<tr>
<td>aes_core</td>
<td>1472</td>
<td>554</td>
<td>17636</td>
<td>169</td>
</tr>
<tr>
<td>ac97_ctrl</td>
<td>4682</td>
<td>2289</td>
<td>21708</td>
<td>42</td>
</tr>
<tr>
<td>dma</td>
<td>5274</td>
<td>2197</td>
<td>27236</td>
<td>287</td>
</tr>
<tr>
<td>wb_conmax</td>
<td>4182</td>
<td>818</td>
<td>34690</td>
<td>99</td>
</tr>
<tr>
<td>pci_bridge32</td>
<td>8166</td>
<td>3677</td>
<td>36647</td>
<td>88</td>
</tr>
<tr>
<td>ethernet</td>
<td>21294</td>
<td>10545</td>
<td>125331</td>
<td>76</td>
</tr>
<tr>
<td>vga_lcd</td>
<td>196</td>
<td>17102</td>
<td>204810</td>
<td>71</td>
</tr>
</tbody>
</table>

differently than Chapter 2). We used a commercial ATPG tool to generate transition-delay fault (TDF) test cubes for these circuits. The ATPG tool was forced to generate launch-on-capture (LOC) transition fault patterns. The primary-input change during capture cycles and the observation of primary outputs was prevented in order to simulate realistic test environments. The path delays were calculated using an in-house dynamic path-timing simulator.

5.2.2 Generating DDPMs for Gate Instances

DDPM of gate instances were generated by running 200 Monte Carlo (MC) simulations on each gate, for all possible input signal transitions. 180nm process technology parameters are used for HSpice simulations. MC simulations were run using the following realistic process-variation parameters for a Gaussian distribution: (i) Transistor gate length $L : 3\sigma = 10\%$; (ii) threshold voltage $V_{TH} : 3\sigma = 30\%$; (iii) gate-oxide thickness $t_{OX} : 3\sigma = 3\%$. We did not model the interconnect delays.
Table 5.2: The comparison of number of excited distinct long paths for a long path limit of 70% of the clock period.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>fill-0</th>
<th>fill-1</th>
<th>rand</th>
<th>dev</th>
<th>[48]</th>
</tr>
</thead>
<tbody>
<tr>
<td>wb_dma</td>
<td>1102</td>
<td>925</td>
<td>2465</td>
<td>3958</td>
<td>5862</td>
</tr>
<tr>
<td>tv80</td>
<td>807</td>
<td>1277</td>
<td>1616</td>
<td>6219</td>
<td>4511</td>
</tr>
<tr>
<td>mem_ctrl</td>
<td>3220</td>
<td>2406</td>
<td>5550</td>
<td>9716</td>
<td>5868</td>
</tr>
<tr>
<td>usb_funct</td>
<td>679</td>
<td>929</td>
<td>1674</td>
<td>3443</td>
<td>2899</td>
</tr>
<tr>
<td>aes_core</td>
<td>57069</td>
<td>58165</td>
<td>86160</td>
<td>123063</td>
<td>85019</td>
</tr>
<tr>
<td>ac97_ctrl</td>
<td>1220</td>
<td>1314</td>
<td>2777</td>
<td>3331</td>
<td>3236</td>
</tr>
<tr>
<td>dma</td>
<td>455</td>
<td>292</td>
<td>827</td>
<td>3899</td>
<td>1262</td>
</tr>
<tr>
<td>wb_conmax</td>
<td>6421</td>
<td>5704</td>
<td>11626</td>
<td>56282</td>
<td>26718</td>
</tr>
<tr>
<td>pci_bridge32</td>
<td>8926</td>
<td>9016</td>
<td>24955</td>
<td>44363</td>
<td>28552</td>
</tr>
<tr>
<td>ethernet</td>
<td>62287</td>
<td>59803</td>
<td>160552</td>
<td>195844</td>
<td>172679</td>
</tr>
<tr>
<td>vga_lcd</td>
<td>36141</td>
<td>36260</td>
<td>48422</td>
<td>49524</td>
<td>48511</td>
</tr>
</tbody>
</table>

Table 5.3: The comparison of number of excited distinct long paths for a long path limit of 80% of the clock period.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>fill-0</th>
<th>fill-1</th>
<th>rand</th>
<th>dev</th>
<th>[48]</th>
</tr>
</thead>
<tbody>
<tr>
<td>wb_dma</td>
<td>0</td>
<td>0</td>
<td>558</td>
<td>1175</td>
<td>1397</td>
</tr>
<tr>
<td>tv80</td>
<td>267</td>
<td>769</td>
<td>783</td>
<td>3802</td>
<td>1737</td>
</tr>
<tr>
<td>mem_ctrl</td>
<td>880</td>
<td>332</td>
<td>1137</td>
<td>3544</td>
<td>891</td>
</tr>
<tr>
<td>usb_funct</td>
<td>490</td>
<td>822</td>
<td>1553</td>
<td>3309</td>
<td>2732</td>
</tr>
<tr>
<td>aes_core</td>
<td>4430</td>
<td>4367</td>
<td>7249</td>
<td>11351</td>
<td>7024</td>
</tr>
<tr>
<td>ac97_ctrl</td>
<td>320</td>
<td>345</td>
<td>500</td>
<td>583</td>
<td>544</td>
</tr>
<tr>
<td>dma</td>
<td>152</td>
<td>93</td>
<td>238</td>
<td>1500</td>
<td>361</td>
</tr>
<tr>
<td>wb_conmax</td>
<td>1622</td>
<td>1339</td>
<td>3530</td>
<td>16507</td>
<td>5899</td>
</tr>
<tr>
<td>pci_bridge32</td>
<td>7422</td>
<td>7985</td>
<td>21939</td>
<td>41578</td>
<td>25512</td>
</tr>
<tr>
<td>ethernet</td>
<td>41172</td>
<td>39736</td>
<td>120449</td>
<td>155960</td>
<td>136368</td>
</tr>
<tr>
<td>vga_lcd</td>
<td>36141</td>
<td>36260</td>
<td>48422</td>
<td>49524</td>
<td>48511</td>
</tr>
</tbody>
</table>

169
Table 5.4: The comparison of number of excited distinct long paths for a long path limit of 90\% of the clock period.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>fill-0</th>
<th>fill-1</th>
<th>rand</th>
<th>dev</th>
<th>[48]</th>
</tr>
</thead>
<tbody>
<tr>
<td>wb_dma</td>
<td>0</td>
<td>0</td>
<td>313</td>
<td>299</td>
<td>577</td>
</tr>
<tr>
<td>tv80</td>
<td>165</td>
<td>356</td>
<td>373</td>
<td>2675</td>
<td>1131</td>
</tr>
<tr>
<td>mem_ctrl</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>usb_funct</td>
<td>12</td>
<td>312</td>
<td>1646</td>
<td>1124</td>
<td></td>
</tr>
<tr>
<td>aes_core</td>
<td>50</td>
<td>190</td>
<td>368</td>
<td>596</td>
<td>308</td>
</tr>
<tr>
<td>ac97_ctrl</td>
<td>92</td>
<td>291</td>
<td>292</td>
<td>292</td>
<td>292</td>
</tr>
<tr>
<td>dma</td>
<td>5</td>
<td>4</td>
<td>26</td>
<td>530</td>
<td>49</td>
</tr>
<tr>
<td>wb_conmax</td>
<td>65</td>
<td>54</td>
<td>86</td>
<td>452</td>
<td>121</td>
</tr>
<tr>
<td>pci_bridge32</td>
<td>3463</td>
<td>3113</td>
<td>10749</td>
<td>24853</td>
<td>12864</td>
</tr>
<tr>
<td>ethernet</td>
<td>9103</td>
<td>8973</td>
<td>29329</td>
<td>41932</td>
<td>34218</td>
</tr>
<tr>
<td>vga_lcd</td>
<td>26629</td>
<td>26678</td>
<td>41354</td>
<td>44202</td>
<td>41984</td>
</tr>
</tbody>
</table>

5.2.3 Results

We configured the size of the LFSR depending on the maximum number of specified bits for each benchmark (Table 2.3). The size of the LFSR is set to $S_{max} + 20$ where $S_{max}$ is the maximum number of specified bits in the test cubes of the corresponding benchmark [74]. This configuration worked for all cases except one of the test cubes of “ethernet”, for which we had to increase the LFSR size to $S_{max} + 26$ in order to find a seed. We removed this test cube from the list of “ethernet” patterns to be consistent with other benchmark configurations.

We set the number of seeds per test cube ($N_s$) to 32 for all benchmarks. These seeds were randomly selected among the candidate seeds for the corresponding test cube. The resulting fully-specified TDF test patterns were used to compute output deviations and to find the most effective seed, as described in Section 5.1. The
size of the effective pattern-list per observation point \( (N_p) \) is set to 5. In addition to output-deviations-based seed selection, we also implemented the filling of unspecified bits of the seeds with “0” (fill-0), with “1” (fill-1), and randomly (random-fill). The difference between random-fill and deviation-based seed selection is that the deviation-based method selects the seed from a pool of randomly filled seeds, whereas random-fill selects the first randomly-filled seed.

The CPU run-time for the various steps are shown in Table 5.5. The first column shows the number of test cubes generated by the ATPG tool. The second column shows the ATPG run time. The third column shows the time needed to find seeds for fill-0, fill-1, and random-fill methods (these methods have approximately the same run-time). The last column shows the total run-time needed for deviation-based method (the generation of 32 seeds for each test cube, deviation computation for all expanded patterns, and pattern sorting). As expected, the deviation-based method takes more CPU time because of the need to run deviation computation on 32x more patterns. Within the deviation-based method’s run time, seed generated takes 5% of the run time, and deviation computation takes 95%. The final pattern-sorting step takes negligible time. The high CPU time is not a serious concern since it is carried out only once during design.

We evaluated the CPU time needed for all the tested methods. As expected, the deviation-based method takes more CPU time because of the need to run deviation computation on 32x more patterns. Within the deviation-based method’s run time, seed generated takes 5% of the run time, and deviation computation takes 95%. The final pattern-sorting step takes negligible time. The high CPU time is not a serious concern since it is carried out only once during design. Furthermore, if multiple
processors are available, deviation-based seed selection method can be spread over any number of processors to decrease the effective time to obtain the results.

Next, we evaluate the long-path excitation capability of the fully-specified patterns for fill-0, fill-1, random-fill, and deviation-based seed selection method. We ran detailed dynamic timing analysis for all the generated patterns in order to find all the excited path delays. Then, we defined different long path limits and counted the number of excited distinct long paths for each case. Tables 5.2-5.4 shows the results of this analysis for a range of long path limits, from 70% of the system clock period to 90% of the period. As seen, deviation-based seed-selection method clearly outperforms fill-0 and fill-1 methods, and does significantly better than random-fill and seeds based on [48]. As we increase the long-path delay limit, we start targeting more low-slack paths. For such paths, the fill-0 and fill-1 methods provide very low coverage compared to the deviation-based method, while the deviation-based method is still considerably better than random-fill.

To evaluate the fault coverage ramp-up provided by all four methods, we ran fault injection simulations. For each benchmark, we inserted 50000 delay defects on randomly chosen nets. We assumed that the additional delay introduced by the injected defects has a distribution of $e^{-Ax}$ as used in [44] and [3]. We let $A = \frac{5}{T_{CLK}}$, where $T_{CLK}$ corresponds to the rated clock period of the circuit under test.

Table 5.6 lists the number of faults detected by each method. The number of detected faults for all the benchmarks is presented as Venn diagrams in Fig. 5.4. We find that the deviation-based method clearly outperforms both random-fill and combined fill-0/1. If we consider test escapes, we see that the deviation-based method missed the least number of faults that would otherwise be detected by other methods.
Figs. 5.5-5.10 shows how the fault coverage increases with the number of patterns for the largest benchmark circuits. Early detection of defects is important, and it can save considerable test time in an abort-on-first-fail methodology. Each plot in these figures show results for the deviation-based method (dev), random-fill (rand), fill-0 and fill-1. We find that the coverage rises more steeply for the proposed method (dev) compared to other methods.

5.3 Summary

We have presented a SDD-aware seed-selection technique for LSFR-reseeding-based test compression. We used the output deviations metric to drive the seed selection algorithm. The selected seeds lead to greater excitation of short-slack paths and achieved significantly better fault detection rates compared to other methods. The proposed method embeds ATPG-generated test cubes for transition-delay faults, hence high transition-fault (TDF) coverage is ensured.
### Table 5.5: Benchmark run-time statistics

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># of test cubes</th>
<th>ATPG time (s)</th>
<th>Basic seed gen. (s)</th>
<th>Dev. based (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac97_ctrl</td>
<td>31,392</td>
<td>561</td>
<td>20</td>
<td>20,790</td>
</tr>
<tr>
<td>aes_core</td>
<td>37,748</td>
<td>732</td>
<td>29</td>
<td>25,628</td>
</tr>
<tr>
<td>dma</td>
<td>47,292</td>
<td>1,279</td>
<td>155</td>
<td>35,827</td>
</tr>
<tr>
<td>ethernet</td>
<td>123,516</td>
<td>11,500</td>
<td>541</td>
<td>406,429</td>
</tr>
<tr>
<td>mem_ctrl</td>
<td>18,975</td>
<td>200</td>
<td>16</td>
<td>5,990</td>
</tr>
<tr>
<td>pci_bridge32</td>
<td>53,921</td>
<td>1,946</td>
<td>91</td>
<td>58,272</td>
</tr>
<tr>
<td>systemcaes</td>
<td>15,806</td>
<td>190</td>
<td>6</td>
<td>4,726</td>
</tr>
<tr>
<td>tv80</td>
<td>12,411</td>
<td>94</td>
<td>4</td>
<td>2,321</td>
</tr>
<tr>
<td>usb_funct</td>
<td>31,544</td>
<td>602</td>
<td>26</td>
<td>16,913</td>
</tr>
<tr>
<td>wb_conmax</td>
<td>99,588</td>
<td>4,237</td>
<td>176</td>
<td>130,194</td>
</tr>
<tr>
<td>wb_dma</td>
<td>9,917</td>
<td>57</td>
<td>4</td>
<td>1,772</td>
</tr>
</tbody>
</table>

### Table 5.6: Fault injection results: Number of detected faults

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Fill-0</th>
<th>Fill-1</th>
<th>Random</th>
<th>Deviation based</th>
<th>Seeds based on [48]</th>
</tr>
</thead>
<tbody>
<tr>
<td>wb_dma</td>
<td>2,322</td>
<td>2,223</td>
<td>3,239</td>
<td>4,014</td>
<td>4,428</td>
</tr>
<tr>
<td>tv80</td>
<td>3,307</td>
<td>3,618</td>
<td>3,983</td>
<td>6,222</td>
<td>5,383</td>
</tr>
<tr>
<td>mem_ctrl</td>
<td>2,712</td>
<td>2,699</td>
<td>3,258</td>
<td>3,795</td>
<td>3,622</td>
</tr>
<tr>
<td>usb_funct</td>
<td>1,657</td>
<td>1,909</td>
<td>2,273</td>
<td>2,930</td>
<td>2,781</td>
</tr>
<tr>
<td>aes_core</td>
<td>10,979</td>
<td>11,002</td>
<td>11,283</td>
<td>11,780</td>
<td>11,243</td>
</tr>
<tr>
<td>ac97_ctrl</td>
<td>4,109</td>
<td>4,140</td>
<td>5,522</td>
<td>6,114</td>
<td>5,934</td>
</tr>
<tr>
<td>dma</td>
<td>1,301</td>
<td>1,145</td>
<td>1,600</td>
<td>2,828</td>
<td>1,921</td>
</tr>
<tr>
<td>wb_conmax</td>
<td>2,018</td>
<td>2,064</td>
<td>2,224</td>
<td>2,518</td>
<td>2,124</td>
</tr>
<tr>
<td>pci_bridge32</td>
<td>2,988</td>
<td>3,003</td>
<td>3,416</td>
<td>3,776</td>
<td>3,395</td>
</tr>
<tr>
<td>ethernet</td>
<td>4,375</td>
<td>4,311</td>
<td>5,994</td>
<td>6,425</td>
<td>6,012</td>
</tr>
<tr>
<td>vga_lcd</td>
<td>9,689</td>
<td>9,734</td>
<td>10,709</td>
<td>11,181</td>
<td>10,888</td>
</tr>
</tbody>
</table>
Figure 5.4: The number of defects detected by the selected patterns for deviation-based (dev), random-fill (rand), and the fill-0 or fill-1 (fill-0/1) methods: (a) wb_dma; (b) tv80; (c) mem_ctrl; (d) usb_funct; (e) aes_core; (f) ac97_ctrl; (g) dma; (h) wb_conmax; (i) pci_bridge32; (j) ethernet; (k) vga_lcd
Figure 5.5: The fault coverage ramp-up using the specified patterns of deviation-based (dev), random-fill (rand), and the fill-0 or fill-1 (fill-0/1) methods: (a) wb_dma; (b) tv80.
Figure 5.6: The fault coverage ramp-up using the specified patterns of deviation-based (dev), random-fill (rand), and the fill-0 or fill-1 (fill-0/1) methods: (a) mem_ctrl; (b) usb_funct.
Figure 5.7: The fault coverage ramp-up using the specified patterns of deviation-based (dev), random-fill (rand), and the fill-0 or fill-1 (fill-0/1) methods: (a) aes_core; (b) ac97_ctrl.
Figure 5.8: The fault coverage ramp-up using the specified patterns of deviation-based (dev), random-fill (rand), and the fill-0 or fill-1 (fill-0/1) methods: (a) dma; (b) wb_conmax.
Figure 5.9: The fault coverage ramp-up using the specified patterns of deviation-based (dev), random-fill (rand), and the fill-0 or fill-1 (fill-0/1) methods: (a) pci_bridge32; (b) ethernet.
Figure 5.10: The fault coverage ramp-up using the specified patterns of deviation-based (dev), random-fill (rand), and the fill-0 or fill-1 (fill-0/1) methods for benchmark vga_lcd.
Chapter 6

Industrial Applications

Commercial timing-aware ATPG tools, such as Mentor Graphics FastScan, have been developed in the recent years as a result of the growing industry concerns regarding SDDs. Furthermore, conflicting market drivers of design cycle time, product quality, selling price and functional density (more functionality per unit area of silicon) require that all aspects of design, manufacture and test are as efficient and cost effective as possible.

Timing-aware ATPG does not come without a cost. Targeted faults require more “care” bits and consequently, more processing time, larger pattern count and more processing resources. The assumed benefits of timing-aware ATPG need to be weighed against the cost and may not be appropriate as a production solution for all designs. Still, there is benefit to the methodology for device speed characterization, performance binning, failure analysis and so forth. For those purposes, the cost is not a recurring part of production overhead. With proper application of timing parameters and a judicious balance between normal transition and timing-aware ATPG patterns, one may achieve a high quality, low cost test for existing and future technologies. To this end we analyzed the performance, efficiency and cost of FastScan timing-aware ATPG against other traditional forms of structural test. Next, we evaluated the deviation-driven pattern selection method on industrial circuits.
The remainder of this chapter is organized as follows. In Section 6.1, we provide a short summary of the background of TA FastScan. Section 6.2 presents a case study of timing-aware ATPG using the FastScan tool, evaluates the timing-aware FastScan on industrial benchmark circuits, and compares it to $n$-detect ATPG. In section 6.3, we present the results of deviation-driven pattern selection method on industrial circuits. Section 6.4 concludes this chapter.

6.1 Background: Timing-aware FastScan

In this section, we discuss the background information regarding timing-aware FastScan. The background algorithm, optimization options, and built-in optimization metrics are discussed.

6.1.1 Algorithm

While targeting SDDs, TA FastScan uses an updated TDF ATPG algorithm as described in [30]. The delay data is extracted from SDF files. Before starting pattern generation, FastScan runs an approximate static-timing analysis step. In this pre-processing step, all static pin delays are obtained. In the pattern generation step, in order to sensitize faults through long paths, each fault is activated and propagated through the longest static paths. However, activation and propagation is done separately, thus sensitizing the absolute longest path passing through a fault is not guaranteed. This imperfection is a trade-off between run-time penalty and the test pattern quality.

If the ATPG engine always selects the longest static paths for each fault, the
same long paths are likely to be exercised over and over again. This results in low
topological coverage of the paths in the DUT. To prevent this scenario, FastScan
uses a weighted randomization scheme during path selection. In this scheme, each
path is assigned a probability of selection that is proportional to the length of the
path. Thus, longer paths are more likely to be selected, but shorter paths are also
not discarded.

6.1.2 Optimization Options

Timing-aware FastScan provides two important optimization options for obtaining
high quality delay test:

- **Critical timing limit**: This is a fault selection criteria that is used before
  running pattern generation. Any fault below the specified critical timing limit
  is dropped from the fault universe. For instance, if the test capture clock period
  is 10ns and the critical timing limit is 80%, any fault with a slack of more than
  2ns is dropped. This optimization option is useful if only longest paths need to
  be targeted by timing-aware ATPG. Furthermore, since pattern generation will
  start with less number of target faults, run-time, pattern count, and memory
  usage is supposed to drop considerably depending on the selection of critical
  timing limit.

  FastScan command: set atpg timing -critical_faults 70%

- **Slack margin (σ)**: ATPG process includes two main steps: Pattern generation
  and fault simulation. Slack margin is a fault dropping criteria during fault
simulation. It determines how successfully the tool should exercise each fault. The definition of slack margin is given in Equation 6.1.

\[
\sigma = \frac{PD^s_f - PD^a_f}{T_{TC} - PD^a_f}
\]  

In Eqn. 6.1, \(PD^s_f\) denotes the longest static path delay passing through a fault location \(f\), \(PD^a_f\) denotes the actual (sensitized) longest path delay passing through the same fault location, \(T_{TC}\) denotes the test capture clock period, and \(\sigma\) denotes the slack margin. As seen, lower slack margin means higher test quality. Assume that the test capture clock period (\(T_{TC}\)) is 10ns, and the longest static path delay passing through the fault location (\(PD^s_f\)) is 8ns. Setting slack margin (\(\sigma\)) to 50% means that FastScan can drop the fault as soon as the actual delay sensitized by the test pattern (\(PD^a_f\)) is 6ns or more. On the other hand, setting slack margin to 0% requires \(PD^a_f\) being 8ns. Thus, if slack margin is set to 0%, FastScan does not drop a fault until it is sensitized through the longest static path or an iteration limit is hit. As a result, CPU time, pattern count, and memory usage increases considerably when high test quality is required. Setting the slack margin to 100% means that the faults can be dropped independent of the sensitized path length, as soon as they are propagated to a scan flop. The default value for the slack margin is 50%.

Note that timing-aware FastScan tries to exercise longer paths independent of the selected slack margin. This is because the underlying algorithm is used in the pattern generation step, even before the slack margin is checked during
fault simulation. However, using a high slack margin may result in many “free
detections”. After a pattern is generated for a fault, many other faults may
be dropped from the fault universe after the fault simulation, because the
required slack margin is satisfied for all of these other faults (free detections).
Considering this fact, setting the slack margin to 100% means that timing-
aware ATPG will behave almost like TDF ATPG, except for a small percentage
of the fault universe.

FastScan command: set atpg timing ON -source SDF
-slack_margin_for_fault_dropping 25%

6.1.3 Delay Test Quality Metrics

Timing-aware FastScan reports two different test quality metrics regarding small-
delay defects: Delay test coverage (DTC) and SDQM. A detailed description SDQM
metric is provided in [3], and the detail description of DTC metric is provided in [30].

6.2 Experimental Results: Timing-aware ATPG
vs $n$-detect TDF ATPG

In this section, we present experimental results obtained for industrial benchmark
circuits for the comparison of timing-aware ATPG and $n$-detect TDF ATPG. We
first provide the details of the experimental setup, followed by a short description of
the benchmark circuits. Next, we present the simulation results.
### Table 6.1: Fault counts for industrial circuit blocks

<table>
<thead>
<tr>
<th>Circuit</th>
<th># of faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circuit1</td>
<td>769,068</td>
</tr>
<tr>
<td>Circuit2</td>
<td>905,908</td>
</tr>
<tr>
<td>Circuit3</td>
<td>1,054,368</td>
</tr>
<tr>
<td>Circuit4</td>
<td>1,317,564</td>
</tr>
<tr>
<td>Circuit5</td>
<td>1,539,220</td>
</tr>
<tr>
<td>Circuit6</td>
<td>1,611,570</td>
</tr>
<tr>
<td>Circuit7</td>
<td>1,641,856</td>
</tr>
</tbody>
</table>

#### 6.2.1 Experimental Setup

All the experiments were performed on a pool of state-of-the-art servers that are all part of a Load Sharing Facility (LSF). Although uniform distribution of hardware configurations (CPU type, total physical memory, memory read/write speed, and similar properties) is not guaranteed, all the experiments were run after reserving at least 12GB of memory. FastScan version v8.2007_3.10 is used for all experiments.

#### 6.2.2 Benchmarks

Blocks were selected from AMD microprocessor designs composed of a combination of standard cells and custom memory macros. Memory macros were timed at the top level only, with internal timing set to zero delay. This is due to the fact that memory macros are large transistor level design blocks. To compensate for macro gate count differences between the design and its test model representation, fault count was used as a measure of circuit size. Table 6.1 shows the number of faults for each block as determined by FastScan.

All standard cells were modeled and timed to match the physical implementation.
Interconnect delays were extracted directly from the physical data. Nominal timing was used for each benchmark.

Clock generation and clock-gater control was provided by a wrapper on each block and controlled through named capture procedures (NCP’s). NCP’s are test timing templates used as a guide by FastScan while generating test patterns. Every ATPG run was performed with NCP’s turned on. All NCP’s were pre-defined by the user. No cases reflect clock control provided by FastScan itself, because the complexity of the microprocessor design requires manual test timing settings. With the use of NCP’s, sequential depth is determined by the number of clock cycles defined by the capture procedure. Different blocks required different depths but all received the full set of NCP’s. As a result, comparisons of processing time as a function of circuit size is not recommended.

Patterns were generated with the \texttt{-auto} switch. No limits were placed on the number of patterns generated nor was any target set for coverage. The tool was allowed to run automatic optimization mode.

### 6.2.3 Simulation Results

The simulations for generating patterns can be grouped into three main categories:

- **n-detect TDF ATPG:** Patterns were generated for a range of multiple-detect values. We used \( n = 1, 3, 5, \) and 8. The focus of these experiments is to compare \( n \)-detect TDF ATPG to timing-aware ATPG in terms of the test quality, CPU time, and resource usage.

- **Timing-aware ATPG using different slack margins:** Timing-aware pat-
terns were generated using a range of slack margins (\(\sigma\)). We ran experiments for \(\sigma = 0\%, 25\%, 50\%, 75\%,\) and \(100\%\). The focus of these experiments is to find the effect of slack margin on test-pattern quality, CPU time, and resource usage.

- **Timing-aware ATPG using different critical timing limits:** Timing-aware patterns were generated using different critical timing limits, i.e., 50\%, 60\%, 70\%, 80\%, and 90\%, in order to find the effect of setting a critical timing on the fault count. In these experiments, we set slack margin to 50\% (default) to isolate the effect of slack margin setting. Our simulations results showed that using a critical timing limit to drop faults before ATPG increased the resource usage of timing-aware ATPG considerable. Further analysis revealed that this optimization option is currently not completely compatible with NCP’s, and it needs to be improved in the newer versions of FastScan. Due to this limitation, we will not provide simulation results for this optimization option.

The most important motivation of timing-aware ATPG is the need for higher quality patterns. We evaluated the pattern quality of \(n\)-detect TDF ATPG and timing-aware (TA) ATPG by using the SDQM and DTC metrics reported by FastScan. The results are shown in Fig. 6.1. All the values shown in Fig. 6.1 are normalized by timing-aware ATPG data with slack margin set to 50\%. For delay test coverage, the higher value means higher quality test pattern set. On the hand, for SDQM, a lower value corresponds to a higher quality test pattern set. As seen in Fig. 6.1, timing-aware ATPG consistently achieves better test quality compared to \(n\)-detect TDF when slack margin is set 25\% or lower. However, the difference between timing-
aware ATPG and $n$-detect TDF ATPG is negligible for comparable values of $n$.

Another important observation in Fig. 6.1 is the high correlation of delay test coverage and SDQM metrics. As seen, delay test coverage results consistently match the SDQM results. This means that the delay test coverage metric can be used instead of the popular SDQM metric when chip failure data is not available, i.e., when test pattern are generated before tape-out.

As shown Fig. 6.1, timing-aware ATPG provides slightly higher quality test patterns for the detection of SDDs if the slack margin ($\sigma$) is set to 0%. It is also valuable to learn the cost of timing-aware ATPG in terms of pattern count, CPU time, and memory usage. Pattern count is a very important factor to keep the test cost within the test budget. If the designs are very large, very high CPU time can also be a
limiting factor by affecting the time-to-market considerations. Although not as important as the first two factors, memory usage can also be concern if the required memory is larger than what is available on the servers.

Tables 6.2.3-6.2.3 summarizes the pattern count and resource usage of \( n \)-detect TDF and timing-aware ATPG. All values are normalized by traditional, 1-detect, TDF ATPG results since it is the comparison point for these aspects. As seen in Table 6.2.3, the cost of timing-aware ATPG can be prohibitive in terms of pattern count when the slack margin is set a low value, which is required for high pattern quality. Up to 15x, and on average 8.3x increase in pattern count is observed when the slack margin is set to 0%. Decreasing the slack margin to 25% cut the pattern count to almost half for most benchmarks, resulting in 3.8x pattern count increase on average. Even the 3.8x pattern count increase compared to TDF ATPG may be unacceptable for most companies. In this case, we conclude that it is necessary to add the capability of sorting timing-aware ATPG patterns to FastScan. After sorting the patterns, the highest quality patterns can be selected easily.

The CPU time (Table 6.2.3) and memory usage (Table 6.2.3) cost of timing-aware ATPG show a similar trend to pattern count results. 14x CPU time increase is observed when the slack margin is set to 0%. If 1-detect TDF pattern generation takes couple of days to run, timing-aware ATPG would take more than a month. Given the time-to-market constraints, spending more than a month only for ATPG may not be feasible.

The memory used by timing-aware ATPG is also very high compared to TDF. Some of this memory is used to store the delay data provided by the SDF file. We can assume that the difference between \( n-1 \) (the first data column in Table 6.2.3) and
Table 6.2: Normalized test cost statistics for $n$-detect TDF and TA ATPG: Pattern count comparison, normalized by $n-1$.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>$n-1$</th>
<th>$n-3$</th>
<th>$n-5$</th>
<th>$n-8$</th>
<th>TA, $\sigma=0%$</th>
<th>TA, $\sigma=25%$</th>
<th>TA, $\sigma=50%$</th>
<th>TA, $\sigma=75%$</th>
<th>TA, $\sigma=100%$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circuit1</td>
<td>1</td>
<td>2.02</td>
<td>3.08</td>
<td>4.75</td>
<td>5.76</td>
<td>2.92</td>
<td>1.97</td>
<td>1.69</td>
<td>1.04</td>
</tr>
<tr>
<td>Circuit2</td>
<td>1</td>
<td>1.96</td>
<td>3.03</td>
<td>4.65</td>
<td>8.87</td>
<td>3.79</td>
<td>2.71</td>
<td>2.44</td>
<td>1.13</td>
</tr>
<tr>
<td>Circuit3</td>
<td>1</td>
<td>1.82</td>
<td>2.54</td>
<td>3.83</td>
<td>9.39</td>
<td>3.2</td>
<td>1.51</td>
<td>1.27</td>
<td>1.04</td>
</tr>
<tr>
<td>Circuit4</td>
<td>1</td>
<td>1.82</td>
<td>2.51</td>
<td>3.58</td>
<td>6.39</td>
<td>5.14</td>
<td>4.92</td>
<td>4.37</td>
<td>1.09</td>
</tr>
<tr>
<td>Circuit5</td>
<td>1</td>
<td>1.44</td>
<td>1.95</td>
<td>3.12</td>
<td>15.08</td>
<td>4.78</td>
<td>2.23</td>
<td>1.49</td>
<td>1</td>
</tr>
<tr>
<td>Circuit6</td>
<td>1</td>
<td>1.62</td>
<td>2.41</td>
<td>3.57</td>
<td>4.65</td>
<td>2.1</td>
<td>1.6</td>
<td>1.52</td>
<td>1.01</td>
</tr>
<tr>
<td>Circuit7</td>
<td>1</td>
<td>1.27</td>
<td>1.79</td>
<td>2.55</td>
<td>8.22</td>
<td>4.89</td>
<td>4.2</td>
<td>3.92</td>
<td>1.05</td>
</tr>
<tr>
<td>Average</td>
<td>1</td>
<td>1.71</td>
<td>2.48</td>
<td>3.72</td>
<td>8.34</td>
<td>3.83</td>
<td>2.73</td>
<td>2.39</td>
<td>1.05</td>
</tr>
</tbody>
</table>
Table 6.3: Normalized test cost statistics for $n$-detect TDF and TA ATPG: CPU time comparison, normalized by $n$-1.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>$n-1$</th>
<th>$n-3$</th>
<th>$n-5$</th>
<th>$n-8$</th>
<th>TA, $\sigma=0%$</th>
<th>TA, $\sigma=25%$</th>
<th>TA, $\sigma=50%$</th>
<th>TA, $\sigma=75%$</th>
<th>TA, $\sigma=100%$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circuit1</td>
<td>1</td>
<td>1.87</td>
<td>3.65</td>
<td>4.35</td>
<td>16.73</td>
<td>5.22</td>
<td>3.89</td>
<td>3.11</td>
<td>1.81</td>
</tr>
<tr>
<td>Circuit2</td>
<td>1</td>
<td>1.77</td>
<td>2.87</td>
<td>3.5</td>
<td>15.76</td>
<td>3.73</td>
<td>2.8</td>
<td>2.21</td>
<td>1.44</td>
</tr>
<tr>
<td>Circuit3</td>
<td>1</td>
<td>1.69</td>
<td>2.29</td>
<td>3.32</td>
<td>12.47</td>
<td>2.33</td>
<td>1.23</td>
<td>1.16</td>
<td>1.01</td>
</tr>
<tr>
<td>Circuit4</td>
<td>1</td>
<td>1.84</td>
<td>3.68</td>
<td>3.88</td>
<td>9.08</td>
<td>7.25</td>
<td>9.12</td>
<td>6.21</td>
<td>2.03</td>
</tr>
<tr>
<td>Circuit5</td>
<td>1</td>
<td>1.48</td>
<td>0</td>
<td>3.1</td>
<td>11.45</td>
<td>2.22</td>
<td>1.62</td>
<td>1.27</td>
<td>1.1</td>
</tr>
<tr>
<td>Circuit6</td>
<td>1</td>
<td>2.13</td>
<td>3.15</td>
<td>4.91</td>
<td>22.34</td>
<td>9.52</td>
<td>5.86</td>
<td>5.76</td>
<td>3.35</td>
</tr>
<tr>
<td>Circuit7</td>
<td>1</td>
<td>3.26</td>
<td>4.15</td>
<td>6.48</td>
<td>10.35</td>
<td>6.03</td>
<td>4.66</td>
<td>4.89</td>
<td>1.45</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>1</strong></td>
<td><strong>2.01</strong></td>
<td><strong>2.83</strong></td>
<td><strong>4.22</strong></td>
<td><strong>14.03</strong></td>
<td><strong>5.19</strong></td>
<td><strong>4.17</strong></td>
<td><strong>3.52</strong></td>
<td><strong>1.74</strong></td>
</tr>
</tbody>
</table>
Table 6.4: Normalized test cost statistics for $n$-detect TDF and TA ATPG: Memory usage comparison, normalized by $n-1$.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>$n-1$</th>
<th>$n-3$</th>
<th>$n-5$</th>
<th>$n-8$</th>
<th>TA, $\sigma=0%$</th>
<th>TA, $\sigma=25%$</th>
<th>TA, $\sigma=50%$</th>
<th>TA, $\sigma=75%$</th>
<th>TA, $\sigma=100%$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circuit1</td>
<td>1</td>
<td>1.02</td>
<td>1.04</td>
<td>1.08</td>
<td>1.9</td>
<td>1.53</td>
<td>1.45</td>
<td>1.43</td>
<td>1.38</td>
</tr>
<tr>
<td>Circuit2</td>
<td>1</td>
<td>1.02</td>
<td>1.05</td>
<td>1.09</td>
<td>5.27</td>
<td>2.02</td>
<td>1.7</td>
<td>1.69</td>
<td>1.45</td>
</tr>
<tr>
<td>Circuit3</td>
<td>1</td>
<td>1.02</td>
<td>1.04</td>
<td>1.07</td>
<td>3.41</td>
<td>1.69</td>
<td>1.53</td>
<td>1.51</td>
<td>1.47</td>
</tr>
<tr>
<td>Circuit4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.01</td>
<td>1.36</td>
<td>1.35</td>
<td>1.35</td>
<td>1.34</td>
<td>1.31</td>
</tr>
<tr>
<td>Circuit5</td>
<td>1</td>
<td>1.02</td>
<td>1.04</td>
<td>1.11</td>
<td>10.56</td>
<td>2.98</td>
<td>1.98</td>
<td>1.71</td>
<td>1.55</td>
</tr>
<tr>
<td>Circuit6</td>
<td>1</td>
<td>1.01</td>
<td>1.02</td>
<td>1.04</td>
<td>1.63</td>
<td>1.4</td>
<td>1.37</td>
<td>1.36</td>
<td>1.3</td>
</tr>
<tr>
<td>Circuit7</td>
<td>1</td>
<td>1.02</td>
<td>1.06</td>
<td>1.11</td>
<td>26.74</td>
<td>11.73</td>
<td>12.05</td>
<td>12.13</td>
<td>2.21</td>
</tr>
<tr>
<td>Average</td>
<td>1</td>
<td>1.02</td>
<td>1.04</td>
<td>1.07</td>
<td><strong>7.27</strong></td>
<td><strong>3.24</strong></td>
<td><strong>3.06</strong></td>
<td><strong>3.02</strong></td>
<td><strong>1.53</strong></td>
</tr>
</tbody>
</table>
the slack margin 100% (the last data column in Table 6.2.3) is the memory used to store the delay data provided by the SDF file. According to this assumption, we can conclude that the portion of the memory used for delay data storage is not the largest contributor. On average, 7x memory usage increase is observed when the slack margin is set to 0%. Given that memory usage is sometimes the limiting factor in ATPG runs, providing 7x more memory for timing-aware ATPG may be not feasible. Note that although it is possible to decrease the memory usage by saving patterns to hard disk (instead of memory) during ATPG, this will considerably increase the CPU time.

For traditional ATPG process, the total CPU time and memory usage per processor can be decreased considerably by using multiple parallel processors. We ran simulations to evaluate the parallel processing performance of timing-aware ATPG. The results can be seen in Fig. 6.2, 1-detect ATPG scales nicely with the number of slave processors, and almost 70% run time can be saved with 8 slave processors. Similarly, 8-detect ATPG scales well with number of processors. The behavior of timing-aware ATPG depends on the slack margin. As slack margin is decreased, the CPU time savings also decrease. For 0% slack margin, no more than 25% CPU time can be saved, and most of this saving is obtained by using 2 slave processors. Using more than 2 slave processors return no meaningful time saving for timing-aware ATPG. In contrast, it has considerable effect for n-detect ATPG. The parallel processing performance of timing-aware ATPG needs to be improved to compensate for the increased CPU time when slack margin is set 0%.
6.2.4 Summary of Timing-aware ATPG Evaluation

Based on the preceding data, it is clear that straight-forward application of timing-aware ATPG may not be appropriate for every design. While each graph shows an incremental improvement in DTC and SDQM, the associated cost is substantial in terms of pattern inflation, processing time and memory usage. The greatest potential contribution to test quality is gained by running timing-aware with slack margin ($\sigma$) set to 0%. This also has the greatest impact in test cost. Pattern count inflates on the order of 8x and processing time by 14x. For designs that already push test volume and/or cannot afford the hit in processing time, the first inclination may be to increase slack margin. However, by comparing the test quality metrics for $n$-detect and slack margin, you will find that there is a very respectable quality correlation between the two. Roughly speaking, $n$-1 performs about the same in SDQM and DTC as $\sigma=100$, $n$-3 about the same as $\sigma=75\%$ and so forth, with less or
similar overhead. The exact $n$-detect value can be selected to match the equivalent $\sigma$ desired. Note that no attempt was made to find an $n$-detect number that produced the same results as $\sigma=0\%$.

Timing-aware ATPG cost can be dropped by pre-screening the fault population and dropping non-timing-critical faults. By doing this, timing-aware ATPG may run faster and the remaining faults, picked up by transition test, will also process faster with no loss of coverage. Another possibility is incorporating a pattern grading function into an integrated TDF/timing-aware flow either within FastScan or driven by an intelligent external pattern grader program.

Production test is not the only application for timing-aware structural test. It has been applied to critical path characterization, failure analysis and fault isolation where test inflation is not an issue and does not impact fixed operating expenses as does production test. However, even in this case, it is desirable to have a method to select the best subset of timing-aware ATPG patterns.

### 6.3 Experimental Results: Application of Deviation-Driven Pattern Selection Method

In this section, we present the application of proposed techniques to industrial circuit blocks. Due to its efficiency in selecting best patterns, we use the pattern grading and selection methods proposed in Chapter 4. Different than the method proposed in Chapter 4, we do not enforce any pattern selection stop criteria, and let the procedure select all the patterns that are considered high quality.
Table 6.5: Gate counts for industrial circuit blocks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Gate Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circuit A</td>
<td>62,103</td>
</tr>
<tr>
<td>Circuit B</td>
<td>39,990</td>
</tr>
<tr>
<td>Circuit C</td>
<td>54,456</td>
</tr>
<tr>
<td>Circuit D</td>
<td>39,944</td>
</tr>
</tbody>
</table>

6.3.1 Experimental Setup

Similar to previous section, all the experiments were performed on a pool of state-of-the-art servers that are all part of a Load Sharing Facility (LSF). Although uniform distribution of hardware configurations (CPU type, total physical memory, memory read/write speed, and similar properties) is not guaranteed, all the experiments were run after reserving at least 16GB of memory. FastScan version v8.2008_4.10 is used for all experiments. The in-house pattern grading and selection tool was coded in C++.

6.3.2 Benchmarks

Blocks were selected from AMD microprocessor designs. Standard cell timing library is used to model nominal delay values and maximum delay variations for each gate instance. Interconnect delays are not modeled. Table 6.5 shows the number of gates for each block as determined by FastScan. Patterns were generated with the -auto switch. No limits were placed on the number of patterns generated nor was any target set for coverage. The tool was allowed to run automatic optimization mode.
6.3.3 Simulation Results

The simulations for generating patterns can be grouped into three main categories:

- **n-detect TDF ATPG**: Patterns were generated for a range of multiple-detect values. We used $n = 1, 3, 5, \text{ and } 8$.

- **Timing-aware ATPG using different slack margins**: Timing-aware patterns were generated using slack margins $\sigma = 0\%$ and $25\%$.

- **Selected Patterns**: We used our in-house pattern grading and selection tool to select high quality patterns from both $n$-detect and timing-aware pattern sets. We used three different path calculation thresholds: 0.7, 0.8, and 0.9, which corresponds to long path limits of 70\%, 80\%, and 90\% of the clock cycle respectively.

Similar to the discussion in earlier chapters, we observed that the pattern count of timing-aware ATPG is much higher than TDF ATPG pattern count. Likewise, as $n$ increases, the number of patterns in the $n$-detect pattern set also increases. Figs. 6.3-6.5 shows the number of test patterns generated by $n$-detect ATPG, timing-aware ATPG, and the number of patterns selected by the proposed scheme. The difference between these figures is the change in the path threshold applied to the proposed scheme. Figs. 6.3-6.5 shows the results for 0.7, 0.8, and 0.9 thresholds, respectively. As seen, for all cases, the number of patterns selected by the proposed method is only a very small fraction of the overall pattern set. When threshold is set to 0.7, the proposed scheme selects only 10\% of the available patterns for Circuit A from the timing-aware pattern set with $\sigma = 0\%$. Similar results are obtained for other
Figure 6.3: Normalized number of test patterns for \( n \)-detect ATPG, timing-aware ATPG (ta), and the proposed pattern selection method (dev). Path threshold is set to 0.7 for the proposed scheme. All values are normalized by the value of \( n-1 \).

benchmarks. As expected, as the threshold increases, the number of selected patterns drop as low as 3\% of the original pattern set.

The results for CPU time usage is as impressive as the pattern count results. Fig. 6.6 shows the normalized CPU time usage results for \( n \)-detect ATPG, timing-aware ATPG (ta), and the proposed pattern grading and pattern selection method. As seen, the complete processing time (pattern grading and pattern selection) for the proposed scheme is only a small fraction of the ATPG run time. For instance, for Circuit C, \( n-8 \) ATPG run time is 10\( x \) longer than the pattern grading and selection time. For Circuit B, the time spent for pattern grading and selection if only 2.5\% of the timing-aware ATPG run time with \( sigma=0\% \).

Since the proposed scheme is let to select as many patterns as needed to cover all high risk paths, the patterns selected by the proposed scheme sensitizes all or nearly
Figure 6.4: Normalized number of test patterns for \( n \)-detect ATPG, timing-aware ATPG (ta), and the proposed pattern selection method (dev). Path threshold is set to 0.8 for the proposed scheme. All values are normalized by the value of \( n-1 \).

all of the long paths that can be excited by the given base pattern set. Figs. 6.7-6.9 shows the results for long path limits 70%, 80%, and 90% respectively. As seen, for all cases, the proposed scheme covered all the long paths that can be sensitized by the given base pattern set, using only a fraction of the test patterns.

The long path coverage ramp up for the selected patterns is also significantly better than both \( n \)-detect and timing-aware ATPG patterns. Figs. 6.10 and 6.11 presents the results for the long-path coverage ramp-up with respect to the number of applied patterns. For all cases, the selected and sorted patterns cover the same number of long paths much faster, and using less number of patterns.
Figure 6.5: Normalized number of test patterns for $n$-detect ATPG, timing-aware ATPG (ta), and the proposed pattern selection method (dev). Path threshold is set to 0.9 for the proposed scheme. All values are normalized by the value of n-1.

6.4 Summary

In this chapter, we presented the application of the techniques proposed in earlier chapters to industrial circuit blocks. Our analysis showed that the results for industrial blocks are inline with the results obtained from academic benchmarks. For both cases, we observed that the cost of currently available commercial timing-aware ATPG is very high in terms of both pattern count inflation and and CPU time increase. $n$-detect ATPG seems to be a good alternative to timing-aware ATPG. However, the base pattern count of $n$-detect ATPG is also high, although it is usually smaller than the timing-aware ATPG pattern count. At this point, the proposed pattern grading and pattern selection techniques can play a vital role. It is shown on industrial circuit blocks that the proposed scheme can obtain the same test quality
Figure 6.6: Normalized CPU time usage for $n$-detect ATPG, timing-aware ATPG (ta), and the proposed pattern grading and pattern selection method (dev). Path threshold is set to 0.8 for the proposed scheme. All values are normalized by the value of $n$-1.

In term of SDD coverage, by using an order of magnitude less pattern count. Since the main motivation of timing-aware ATPG is SDD detection, the same quality can be obtained by using significantly less resources. Top-off TDF ATPG can be run to compensate for the lost TDF coverage.
Figure 6.7: Normalized number of sensitized distinct long paths for \( n \)-detect ATPG, timing-aware ATPG (ta), and the proposed pattern selection method (dev). Path threshold is set to 0.7 for the proposed scheme. All values are normalized by the value of \( n-1 \).

Figure 6.8: Normalized number of sensitized distinct long paths for \( n \)-detect ATPG, timing-aware ATPG (ta), and the proposed pattern selection method (dev). Path threshold is set to 0.8 for the proposed scheme. All values are normalized by the value of \( n-1 \).
Figure 6.9: Normalized number of sensitized distinct long paths for $n$-detect ATPG, timing-aware ATPG (ta), and the proposed pattern selection method (dev). Path threshold is set to 0.9 for the proposed scheme. All values are normalized by the value of $n-1$. 
Figure 6.10: The long-path coverage ramp-up using the base 8-detect TDF ATPG (n-8(base)), timing-aware ATPG with $\sigma=25\%$ (ta,s25(base)) and $\sigma=0\%$ (ta,s0(base)), selected patterns from 8-detect TDF ATPG (n-8(selected)) and timing-aware ATPG (ta,s0(selected)). Long path limit is set to 80%: (a) Circuit A; (b) Circuit B.
Figure 6.11: The long-path coverage ramp-up using the base 8-detect TDF ATPG (n-8(base)), timing-aware ATPG with $\sigma=25\%$ (ta,s25(base)) and $\sigma=0\%$ (ta,s0(base)), selected patterns from 8-detect TDF ATPG (n-8(selected)) and timing-aware ATPG (ta,s0(selected)). (a) Circuit C, long path limit is set to 80%; (b) Circuit C, long path limit is set to 90%.
Chapter 7

Conclusions and Future Work

Timing-related defects are major contributors to test escapes and in-field reliability problems for today’s very deep submicron integrated circuits. Small-delay variations induced by crosstalk, process variations, as well as resistive opens and shorts, can potentially cause timing failures in a design, thereby leading to quality and reliability concerns. The research reported in this dissertation presents a test-grading technique that uses the method of output deviations for screening small-delay defects (SDDs). The goal of this research is to provide efficient and effective test-pattern grading and pattern-selection techniques to target SDDs.

In this thesis, a new layout-aware delay defect probability measure is defined to model delay variations for nanometer technologies. The proposed technique intelligently selects the best set of patterns for SDD detection from an $n$-detect pattern set generated using timing-unaware automatic test-pattern generation (ATPG). It offers significantly lower computational complexity and it excites a larger number of long paths compared to todays commercial timing-aware ATPG tools. Most of the SDDs inducing effects are covered in this research. The thesis research also covers pattern compression techniques that are aware of SDDs. The applicability of proposed methods are demonstrated on industrial circuits.
Chapter 2 presented a test-grading technique, based on output deviations, for screening small-delay defects (SDDs). The concept of output deviations was defined for pattern-pairs and shown that it can be used as an efficient surrogate metric to model the effectiveness of transition delay-fault (TDF) patterns for SDDs. A gate-delay defect probability measure is introduced to model gate delay variations for nanometer technologies.

Chapter 3 extended the concepts introduced in Chapter 2 to cover interconnect related process variations. A layout-aware pattern-selection technique was presented for screening SDDs. Several techniques were proposed to consider delay variations on interconnects at different levels of detail. Proposed techniques were compared to each other, as well as today’s commercial timing-aware ATPG tools.

Chapter 4 introduces new techniques to cover the effects of crosstalk and to overcome the shortcomings of the deviation propagation method. A new pattern selection technique is presented. The proposed technique was shown to clearly outperform commercial timing-aware ATPG tools in both the pattern quality and the small number of test patterns.

Chapter 5 presented a SDD-aware seed-selection technique for LSFR-reseeding-based test compression. The output deviations metric was used to drive the seed selection algorithm. The selected seeds lead to greater excitation of short-slack paths and achieved significantly better fault detection rates compared to other methods. The proposed method showed that when LFSR seeds are selected carefully to detect SDDs, the pattern quality can be improved significantly over a random seed selection method.
method.

Chapter 6 showed the applications of methods presented in previous chapters to some representative industrial circuits. The applicability of the proposed methods to real life circuits were clearly demonstrated in this chapter. Our analysis showed that the proposed techniques are not only applicable to academic benchmarks, but they can also be used on industrial circuits.

7.2 Future Work

This thesis explored test pattern grading and pattern selection techniques for the detection of SDDs. In the proposed techniques, process variations, crosstalk, as well as resistive shorts and opens were considered. As the technology scales down to even deeper submicron levels, new test challenges will appear. We next summarize possible future research directives.

7.2.1 Consideration of Power-Supply Noise and Other SDD-Inducing Effects

Although this research work covered most of the SDD inducing effects, it did not cover all of them. Power-supply noise is one example. With slight improvements, the proposed techniques can be used to account for other SDD inducing effects.
7.2.2 Power-aware vs. Layout-aware Test-Pattern Grading and Pattern Selection

In this thesis, we strictly targeted sensitizing paths with SDDs. The notion of power-awareness was not considered. Today’s new design and test challenges requires new test techniques to be power-aware in addition to layout-aware. Test patterns causing a very large signal switching activity can create significant power supply noise and clock stretching. As a result, good parts may fail in production test although they can work without any problem in the field. This situation is also known as over-testing.

Selecting power-aware test patterns and targeting SDDs at the same time is a difficult problem. This is mainly due to the conflicting nature of the two objectives. Power-aware test should minimize the switching activity in the circuit, whereas SDD-aware test needs to target as many long paths as possible for an effective and efficient test. A possible solution should be aware of the functional switching activity of the circuit under test, and adjust the switching activity of the SDD-aware patterns to match functional activity. This technique of test selection can potentially solve both problems. However, this method may also lead to increased pattern count. Careful analysis is needed to find an optimum solution to this problem.

7.2.3 Deviation-based Transition Delay-Fault ATPG

This research targeted detecting SDDs by selecting a small, high quality, and efficient subset of patterns from a large pattern repository. Without loss of generality, \( n \)-detect TDF ATPG was used as the main pattern source. Due to the long pattern generation time, \( n \)-detect ATPG is not the optimal solution. The major contributor
to the overall run time was shown to be $n$-detect TDF pattern generation step. The overall run time can be dropped significantly if this initial step is eliminated.

There are several possible solutions to this problem. First one is to use pseudo-random patterns as the pattern source. Pseudo-random patterns can be used intelligently if the pattern generation step is guided by a genetic algorithm. Deviation-based pattern grading method can be used as a fitness function. In this procedure, previously found high quality patterns can be used to generate new high quality patterns by passing them through a mutation process. If it is shown that a high quality pattern can be transformed into another high quality pattern by switching a few bits, this method can significantly reduce run time. It may even open a door to new ATPG techniques.

A second possible solution is to integrate the deviation-based procedure into an ATPG tool. This will potentially eliminate the time wasted to generate low quality patterns. This solution can be merged with the first proposal to develop a novel ATPG technique that is aware of all SDD-inducing effects.
Appendix A

Example DDPM Tables

In this chapter, real DDPM entries for a range of randomly selected instances are provided. 180nm process technology parameters are used to extract these results from Monte Carlo simulations. All of the instances listed below are selected from the IWLS2005 benchmark aes_core.

Table A.1: DDPM for a 2-input OR gate instance

<table>
<thead>
<tr>
<th>OR</th>
<th>Initial Input State</th>
</tr>
</thead>
<tbody>
<tr>
<td>prob</td>
<td>00 01 10 11</td>
</tr>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>IN0</td>
<td>0.102 0 0.105 0.101</td>
</tr>
<tr>
<td>IN1</td>
<td>0.046 0.048 0 0.101</td>
</tr>
</tbody>
</table>

Table A.1 shows the DDPM table for an OR2 instance. Note that the values given in Table A.1 are much smaller than the illustrative example in Table 2.1. Similarly, Tables A.2-A.4 can be compared to Table 2.2. Note that DDPM entries can sometimes be very close 0 if the expected transition has very small delay defect probability. Tables A.3 (line $\text{IN}_1$) and A.4 (line Inst3) have examples of this situation.

Table A.2: DDPM for a 2-input AND gate instance

<table>
<thead>
<tr>
<th>AND</th>
<th>Initial Input State</th>
</tr>
</thead>
<tbody>
<tr>
<td>prob</td>
<td>00 01 10 11</td>
</tr>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>IN0</td>
<td>0.091 0.005 0 0.005</td>
</tr>
<tr>
<td>IN1</td>
<td>0 0.004 0.004 0.004</td>
</tr>
</tbody>
</table>

213
Table A.3: DDPM for a 2-input XOR gate instance

<table>
<thead>
<tr>
<th>XOR</th>
<th>Initial Input State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>prob</td>
</tr>
<tr>
<td>Inputs</td>
<td>00</td>
</tr>
<tr>
<td>IN0</td>
<td>0.065</td>
</tr>
<tr>
<td>IN1</td>
<td>0.000</td>
</tr>
</tbody>
</table>

Table A.4: DDPM for three different INVERTER gate instance

<table>
<thead>
<tr>
<th>INV</th>
<th>Initial Input State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>prob</td>
</tr>
<tr>
<td>Inst1</td>
<td>IN0</td>
</tr>
<tr>
<td>Inst2</td>
<td>IN0</td>
</tr>
<tr>
<td>Inst3</td>
<td>IN0</td>
</tr>
</tbody>
</table>
Bibliography


Biography

Mahmut Yilmaz

my@ee.duke.edu

PERSONAL DATA
Date of birth: March 15, 1981.
Place of birth: Mersin, Turkey.

EDUCATION
Doctor of Philosophy, Duke University, USA, expected 2009.
Master of Science, Duke University, USA, 2006.
Bachelor of Science, Bogazici University, Turkey, 2004.

PUBLICATIONS
• Refereed Conference Papers


• Submitted Papers


PROFESSIONAL ACTIVITIES

• IEEE student member

• Served as a reviewer for the IEEE Transactions on VLSI, IEEE Transactions on CAD and the International Test Conference.

WORK EXPERIENCE

• AMD, Sunnyvale, CA – DFT Group – Design Engineer 2, Oct 2007 - Present


• Siemens, Istanbul, Turkey – Programmer, Jul 2001 - Jul 2002

*Not related to Ph.D. thesis work.