國立臺灣大學電機資訊學院電子工程學研究所
博士論文

Graduate Institute of Electronics Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Doctoral Dissertation

利用圖形處理器高品質測試向量產生器
GPU-Based High Quality ATPG

廖官榆
Liao, Kuan-Yu

指導教授：李建模博士
Advisor: Li, Chien-Mo, Ph.D.

中華民國 103 年 6 月
June, 2014

國立臺灣大學博士學位論文
口試委員會審定書

利用圖形處理器高品質自動測試向量產生器
GPU-Based High Quality ATPG

本論文係廖官榆君（F97943076）在國立臺灣大學電子工程學研究所完成之博士學位論文，於民國 103 年 6 月 20 日承下列考試委員審查通過及口試及格，特此證明

口試委員： 李建模
（指導教授）
[illegible] 饒建奇
[illegible] 劉靖家

系主任、所長 劉深淵

GPU-Based High Quality ATPG

By

Liao, Kuan-Yu

Doctoral Dissertation

Submitted in partial fulfillment of the requirement
for the degree of Doctor of Philosophy
in Graduate Institute of Electronics Engineering
at National Taiwan University
Taipei, Taiwan, R.O.C.

June 2014

Approved by：

[illegible] [illegible] [illegible]

[illegible] Chien-Mo Li

Advised by： Chien-Mo Li

Approved by Director： [illegible]

中文摘要

由於製程的演進，晶片測試面臨兩個重要的議題：複雜的缺陷表現行為以及自動測試向量產生器 (ATPG) 過於耗時。針對這兩個議題，我們提出利用圖形處理器 (GPU) 的高品質自動測試向量產生器的演算法。傳統利用中央處理器 (CPU) 的自動測試向量產生器通常仰賴非常快速的選擇，以及一次只針對一個錯誤產生一組測試向量。我們提出的自動測試向量產生器不同於傳統，可以同時針對多個錯誤以及多重的測試目標產生測試向量。我們所提出的方法實做了三個層面的平行化：裝置層面 (device-level) 的錯誤分區平行化、區塊層面 (block-level) 的電路分區平行化、以及字組層面 (word-level) 的搜尋分區平行化。這是一個可以同時產生上千個測試向量高度平行化的演算法。這個演算法的核心是一個「分裂成 W 個分身」(Split-into-W-Clones) 的自動測試向量產生器。這個演算法把產生測試向量過程需要做的選擇轉換成平行的字組邏輯運算，而使得許多的選擇可以同時進行。我們更進一步利用圖形處理器來加速這個演算法。我們也提出三個延伸的應用來解決時序感知 (timing-aware) 與標準元件感知 (cell-aware) 相關的議題。實驗結果證明我們提出的演算法不論在品質、運算時間、以及測試向量長度大多優於目前業界的利用中央處理器的自動測試向量產生器。

關鍵字：圖形處理器、測試品質、自動測試向量產生器、平行計算

Abstract

Due to the scaling of the manufacturing technology, two issues are critical in testing modern chips: 1) complex defect behavior; and 2) long automatic test pattern generator (ATPG) runtime. To deal with these issues, a graphical processing unit (GPU) based ATPG framework is proposed. Unlike central processing unit (CPU) based ATPG, which relies on fast serial decision making and generates one test pattern at a time, the proposed framework is capable of targeting multiple test objectives and multiple faults at the same time. This framework provides a completely new approach to the current test issues. The framework implements three levels of parallelisms: *device*-level fault partitioning, *block*-level circuit partitioning, and *word*-level search space partitioning. The result is a massively paralleled algorithm which can generate thousands of patterns simultaneously. Such parallelism has not been achieved on traditional CPU-based ATPG. The core of the framework is the *Split-into-W-Clones* (SWK) parallel ATPG algorithm, which can generate test patterns that meet multiple objectives. SWK uses *random split* to convert decisions into parallel bitwise logic operations so that multiple objectives can be tried at the same time. A GPU-based massively parallel technique is then proposed to accelerate SWK algorithm. Three extensions are also prospoed based on the framework to deal with timing-aware and cell-aware issues. Results show that the framework provides higher quality, shorter test length, and shorter runtime compared with state-of-the-art CPU-based commercial ATPG.

Key words: GPU, test quality, ATPG, parallel computing

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Due to the increasing investment required to produce and apply functional test patterns, structural test is gaining popularity since 1970s [Meuhldorf 76]. Single stuck-at fault has been a popular fault model widely used in both academic research and industry [Roth 66]. As popular as stuck-at fault model, early experiments point out that functional tests can still capture unique escapes that are not detected by higher single stuck-at fault coverage structural test patterns [Maxwell 92]. The Murphy chip experiment also showed that 100% stuck-at fault coverage does not guarantee detecting all defective chips [Ma 95]. Interestingly though, stuck-at 5-detect test pattern was also applied to the experiment and detected some chips that were not detected by the single detect test set. This clearly indicates that more complex fault models are needed. More recent studies also supports the claim: only 5% of the detected defects behaved like stuck-at faults [Mitra 04].

Over the years, many test metrics have been proposed to improve the stuck-at model, including N-detect [McCluskey 00], transistor stuck-on and stuck-off [Wadsack 78], interconnect bridging [Friedman 74], IDDQ [Malaiya 82], gate exhaustive [Cho 05], transition delay [Waicukauski 87], path delay [Smith 85], delay test coverage [Lin 06], small delay quality level [Sato 05], physical-aware N-detect [Lin 08], cell-internal defects [Hapke 09], etc. All these test metrics serve the purpose of guiding ATPG to generate test patterns that detect complex defects. However, there is no single test metric that guarantees to screen out all defects. One study showed that IDDQ tests identifies defects that is not detected by stuck-at and transition delay model [Maxwell 91]. Another study showed that cell-internal defect modeling detects defects that were not detectable by gate exhaustive test patterns [Hapke 09]. Still another study showed that timing tests detect defects that passed all other tests [Nigh 97]. In conclusion, we should not rely upon one single test metric to achieve the maximum test effectiveness.

Along with test metrics, automatic test pattern generator (ATPG) is improving consistently over the past 40 years since the introduction of D Algorithm [Roth 66]. PODEM was proposed to reduce search space size [Goel 81]. FAN provides unique sensitization and multiple backtraces [Fujiwara 83]. TOPS introduced dominators [Kirkland 87]. SOCRATES speed up the test generation considerably by using static and dynamic learning [Schulz 88]. ATOM further optimizes PODEM to reduce runtime [Hamzaoglu 98]. The advances in ATPG algorithms impacts the runtime up to 25,057x speed up [Rearick 06]. However, the runtime speedup is measured with respect to stuck-at fault test generation. When dealing with complex test metrics, the runtime is still an issue. Cell-aware ATPGs, such as gate exhaustive ATPG, which exhaustively tests all gate input combinations, need more than hundred times the runtime of single stuck-at fault ATPG [Ma 95]. Another cell-aware ATPG, which exhaustive simulate cell-internal defects activation conditions, need to perform SPICE simulation on all library cells [Hapke 09]. Timing-aware ATPGs, which activates and propagates transition delay through long paths, need more than 209x runtime of single transition fault ATPG [Yilmaz 08a].

This dissertation offers a innovative ATPG framework to solve the complex defect behavior and the runtime issues. Completely different from traditional ATPG concept, which replies on a powerful processing unit to make smart decisions (through learning, testability, etc.), the proposed framework generates multiple test patterns targeting multiple test metrics and different faults simultaneously. The concept of the proposed framework is to generate a lot of high quality test patterns very quickly, and then produce a compact test set through test selection. The core of proposed framework is a word-level parallel ATPG algorithm capable of targeting multiple test objectives. Then, GPU accelerates the core algorithm through three levels of parallelism. Experiments on advanced quality metrics test generation showed that the proposed framework provides high quality test patterns in short runtime. In conclusion, the proposed GPU-based ATPG framework offers a new perspective to the current test generation approaches.

1.1 Motivation

Due to the scaling of the manufacturing technology, two issues are critical in testing modern chips: 1) complex defect behavior; and 2) long ATPG runtime. Complex defect behavior often requires complex or multiple test metrics to measure the test quality. Since the test metrics are complex, traditional ATPGs require long runtime to generate high quality test patterns. So the challenge of current IC testing is quickly generates high quality test patterns targeting complex or multiple test metrics. The following subsections describe complex defect behaviors and ATPG runtime for targeting complex test metrics.

1.1.1 Test solutions for complex defect behaviors

It has been shown that traditional single stuck-at fault test patterns are not effective enough for complex defect behavior in advanced technology [Ma 95]. To deal with complex defect behaviors, various test solutions have been proposed. The test solutions can be categorized into four categories: 1) test metrics for improving untargeted defect coverage; 2) small delay test metrics and timing-aware ATPG; 2) intra-cell test metrics and cell-aware ATPG; and 4) targeting multiple test objectives simultaneously.

Since it is not possible to generate test patterns that considers all fault models, test metrics that improve untargeted defect coverage such as N-detect has been evaluated in [Venkataraman 04]. In an N-detect test set, each fault is detected by at least N different patterns. It has been shown that N-detect test sets improve untargeted defect coverage and are more effective than single detect test sets [Mitra 04]. In addition to applying N-detect directly in production test, many researches shown that N-detect test set is also suitable for test selection for other test metrics [Peng 10]. Traditional N-detect ATPG is often implemented by running single fault ATPG multiple times while propagating faulty effect from easiest N paths. This is time-consuming and results in low diversity in test patterns. Thus, it is important for N-detect ATPG to generate diversified test patterns efficiently.

Small delay defects (SDD) are gaining more and more attention in modern nanometer technologies and requires timing-aware solutions [Sato 05]. The cause of the extra small delay varies, such as crosstalk, IR-drop, resistive open, etc. Fig. 1.1 shows a resistive open wire that would cause extra small delays [Jahangiri 05]. Delay test coverage (DTC) has been proposed to calculate the effectiveness of a test set in detecting transition faults through long paths [Mitra 04]. Delay slack margin (DSM) was proposed to determine fault dropping threshold for test patterns that detects faults through long enough paths [Lin 06]. Another test metric, small delay quality level (SDQL), has been proposed to provide a quantitative measure on the delay test quality given a delay defect distribution. Timing-aware ATPG [Gupta 04], which tries to activate and propagate fault effects through long paths, is a solution to addressing the SDD issue. However, it has been shown that the runtime and test length of timing-aware ATPG are much longer than those of the timing-unaware ATPG [Peng 10]. An alternative approach is N-detect transition fault test generation followed by test selection [Lee 06]. In such an approach, each fault is detected via N different paths, and then test patterns that detect faults through long paths are selected. Although timing-unaware N-detect ATPG is faster than timing-aware ATPG, the test quality is often lower when many faults have more than N sensitizable paths since the ATPG usually targets short paths first. In order to achieve good test quality, N is typically very large, which will result in extra runtime and longer test length. The solution to the runtime, test length, and test quality issues is either having a fast and compact timing-aware ATPG, or a fast and highly diversified N-detect ATPG with test selection that maintains test quality.

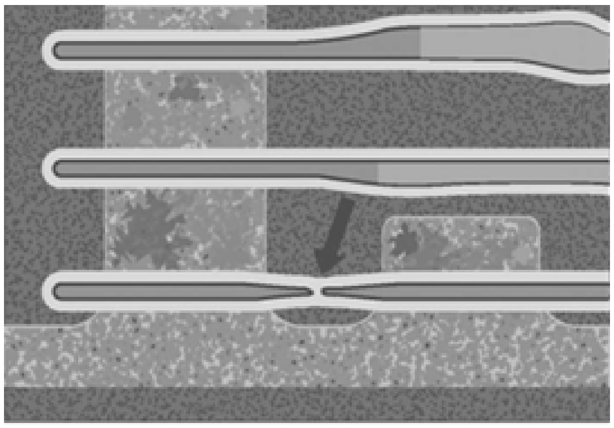
Figure 1.1: SDD caused by resistive open wire.

Intra-cell defects such as cell internal via open, cell internal bridge, etc. require cell-aware solutions. Fig. 1.2 shows an example of cell internal via open [Hapke 09]. Intra-cell defects often require certain input combinations to activate the faulty effect. To address the issue of intra-cell defects, gate exhaustive (GE) ATPG is proposed in [Cho 05]. The gate exhaustive ATPG tries to apply all possible input combinations to each gate. Another approach is SPICE cell-aware ATPG [Hapke 09], which first injects defects to each cell, and then performs SPICE simulation to identify activation conditions. The SPICE cell-aware approach is capable of generating single capture or double capture at speed test patterns to detect intra-cell defects. However, according to our research, some defects in fin shaped field effect transistor (FinFET) cause small extra gate delay that depends on how the transition is launched. This means, the ATPG will need to excite the faulty effect through specific cell input transition combination and try to propagate the faulty effect through long path. Currently, there is no test metric or ATPG to handle this problem.

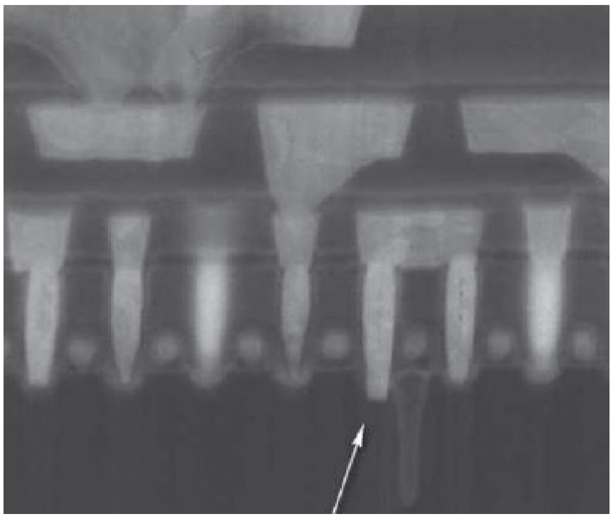

Figure 1.2: Intra-cell via open.

In practical cases, multiple test objectives may need to be satisfied [Nigh 97]. For example, to handle both intra-cell and inter-cell defects, ATPG might need to simultaneously target GE and physical-aware N-detect (PAN) [Lin 08]. To target multiple test objectives, traditional ATPG adds more objectives to stuck-at fault test generation engine [Lee 02], which is time-consuming since additional objectives may conflict with each other. Another approach is using top-off method to target multiple test objectives incrementally [Alampally 11]. However, there are two drawbacks using the top-off approach. First, top-off method is unable to handle multiple test objectives simultaneously. Second, test patterns that are generated targeting latter test objectives might make those targeting the former test objective redundant. To solve this problem, an ATPG that targets multiple test objectives is required.

In conclusion, a multiple objective ATPG framework is needed to provide N-detect, timing-aware, and cell-aware solutions.

1.1.2 ATPG runtime for targeting complex test metrics

Generating test patterns targeting complex test metrics is time-consuming. Take transition fault N-detect for example, in our experiment,it took more than 10 days for a CPU-based single-core commercial tool to generate an 8-detect transition fault test set for a 1.1M gate circuit. As for the timing-aware ATPG, Fig. 1.3 shows the runtime for generating timing-aware test patterns relative to that of generating single detect transition fault test patterns [Yilmaz 08b]. From the figure, it can be seen that the relative runtime can be as high as 209x when the number of faults is large.

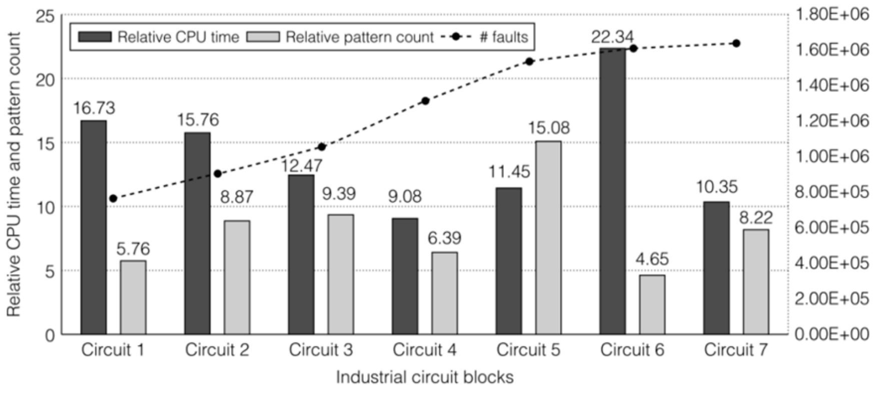

Figure 1.3: Commercial timing-aware ATPG runtime.

To reduce the runtime, many CPU-based parallel ATPG algorithms have been proposed. Parallel ATPG algorithms can be divided into several categories: fault parallelism, heuristic parallelism, search space parallelism, functional parallelism, circuit partitioning, and simulation parallelism [Klenke 92]. However, even in the multicore environment, it still took more than 16 hours for a CPU-based commercial ATPG to generate timing-aware test patterns for a 240K circuits using ten CPU cores. Fig. 1.4 shows the runtime of CPU-based timing-aware commercial ATPG with multicore support on the benchmark circuit b19. It can also be seen from the figure that the speedup saturates when using more than six cores.

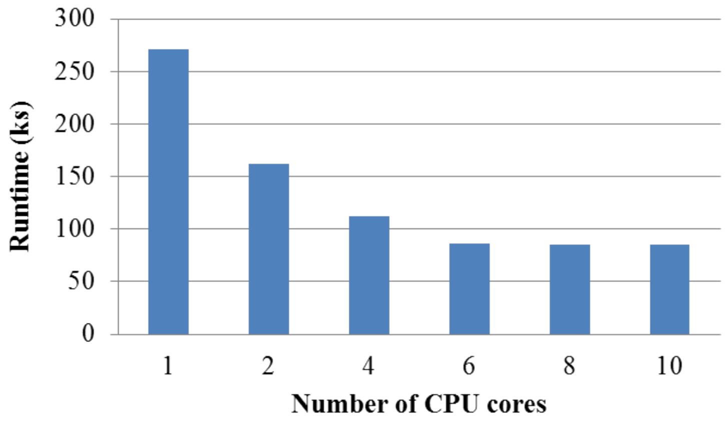

Figure 1.4: CPU-based timing-aware multicore commercial ATPG runtime.

GPU offers another alternative to powerful parallel computing and is gaining popularity in various EDA applications. Fig. 1.5 shows the computation throughput comparison between GPU and CPU. As can be seen in the figure, the number of giga floating point operations (GFLOP) per second of the GPU device is nearly five times to that of current CPU processors. Modern GPUs contain several stream multiprocessors, each of which executes a *thread block* (or a *block* for short). Many researchers proposed GPU-based logic simulation and fault simulation techniques in recent years. Based on the partitioning schemes, simulation techniques can be classified into three categories: fault partitioning [Li 10], pattern partitioning [Gulati 08], and circuit partitioning [Kochte 10]. Different partitions are assigned to different blocks for parallelization. All approaches above can be easily combined with word-level parallel techniques, such as well-known parallel fault simulation [Seshu 65] and parallel pattern single fault propagation (PPSFP) [Waicukauski 89], which are widely used on CPU. So far there is no efficient GPU-based ATPG algorithm available yet.

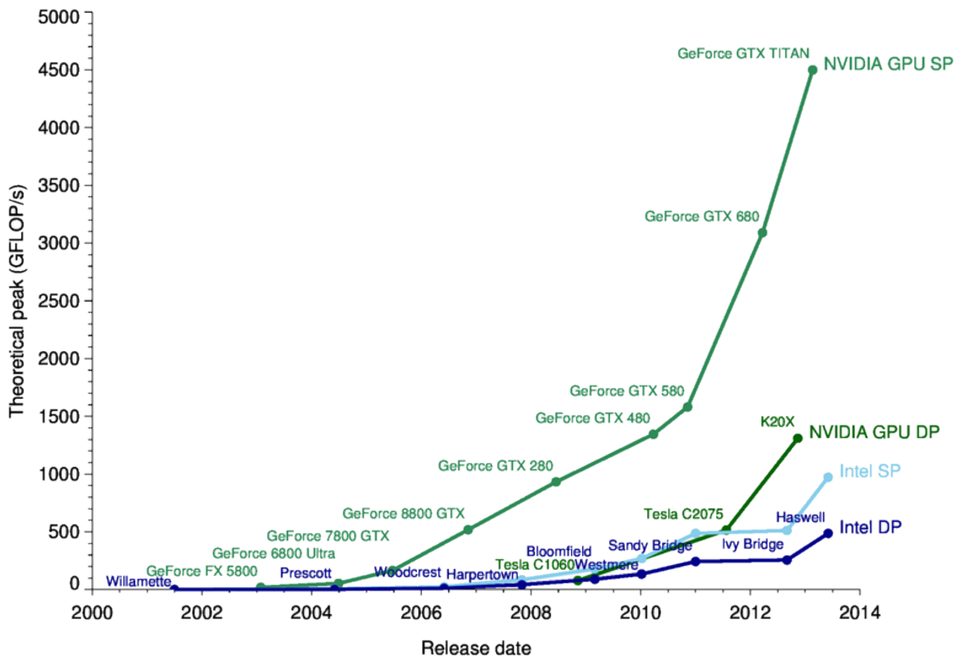

Figure 1.5: Comparison of GPU and CPU computation throughput.

1.2 Goal

The goal of this work is to solve the aforementioned defect behavior and runtime issues. The solution is a GPU-based ATPG framework, which is capable of generating high quality test patterns in a short time while targeting multiple test objectives. Fig. 1.6 shows the proposed framework. The center of the figure is the two key components of this framework, the *Split-into-W-Clones* (SWK) algorithm and the GPU parallelization. The four outer boxes show that the framework offers layout-aware, cell-aware, timing-aware, and multiple test objectives solutions.

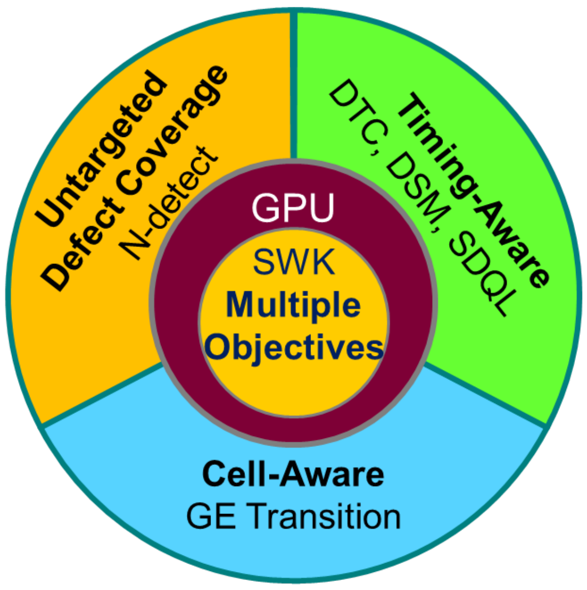

Figure 1.6: The proposed ATPG framework.

The core of this framework is the *Split-into-W-Clones* (SWK) algorithm [Liao 11]. The name of the algorithm is inspired by the fictional heroic money named “Sun Wu Kong”, who can magically clone himself to gain advantages in combat. SWK is a word-level parallel ATPG algorithm where each bit in a word is called a *clone* and each *clone* explores the search space independently. Unlike traditional ATPG which relies on powerful decision making to justify objectives, SWK converts decision making into parallel bitwise logic operation so multiple decisions can be searched at the same time. When a two-way decision is needed, the SWK algorithm randomly splits into two groups of *clones*, each of which performs its own independent search. In other words, SWK is capable of making multiple decisions through performing same bitwise logic operations. This makes SWK suitable for single instruction multiple threads (SIMT) parallel architecture, such as GPU, where each thread performs same operations. Another key feature of SWK is that each clone can have multiple objectives at a time, and different clones can have different objectives. For example, the first clone can target fault f_1 and test metric PAN, whereas the second clone can target fault f_2 and test metric GE. To show that SWK can handle multiple objectives, test patterns are generated targeting stuck-at fault N-detect, PAN, GE, and BCE. The results show that the proposed ATPG produces higher quality test patterns compared to commercial tool 50-detect test set. In conclusion, SWK is a word-level parallel algorithm that targets multiple test metric objectives and is suitable for GPU parallelization.

The GPU parallelization exploits three levels of parallelism: *word*-level search space partitioning, *block*-level circuit partitioning, and *device*-level fault partitioning [Liao 13]. Word-level search space partitioning implements the SWK algorithm, which searches different parts of decision trees using different clones. Block-level circuit partitioning assigns different logic levels to different blocks. Device-level fault partitioning statically partitions faults into different fault lists, each of which is assigned to a different GPU device. In logic and fault simulation, two-level parallelism is implemented: block-level circuit partitioning and word-level pattern partitioning. Block-level circuit partition assigns different logic levels to different blocks, whereas word-level pattern partitioning assigns different patterns to different bits. Suppose the number of devices is v, the number of blocks is t, and the size of a word is w. Our ATPG generates $v \times t \times w$ patterns concurrently. For example, if $v = 2$, $t = 64$, and $w = 32$, then 4,096 patterns can be generated simultaneously. The logic and fault simulation are also implemented on GPU for fault dropping. Both simulations implement two-level parallelism: block-level circuit partitioning and word-level pattern partitioning. Block-level circuit partition assigns different logic levels to different blocks, whereas word-level pattern partitioning assigns different patterns to different bits. The proposed GPU ATPG framework is suitable for generating N-detect test patterns. Each fault can be targeted multiple times by different bits in a word and different faults can be targeted by different blocks simultaneously. The proposed ATPG is compared with CPU-based multicore commercial ATPG in generating transition fault N-detect test patterns. The results show that the proposed ATPG is 1.6 times faster than an 8-core CPU-based commercial ATPG. In summary, this is a massively parallel ATPG that cannot be achieved on traditional CPU architecture.

To deal with SDDs, the proposed framework provides two approaches: 1) fast N-detect test patterns generation with test selection [Chang 13]; and 2) fast high quality timing-aware test generation [Liao 14]. The first approach implements a novel GPU-based test selection algorithm to select high quality test patterns quickly. The test selection algorithm uses static upper and lower bound analysis to quickly decide whether a particular pattern drops a fault or not. If the static bounds meets certain conditions, then the fault can be dropped without detailed path length calculation. Then, a partial fault dictionary is build to only record those faults that have not been dropped before (likely to be ”hard to detect faults”) so that the dictionary size is very small. DSM is used to determine the fault dropping threshold and DTC is used to mearsure the quality of the selected test set. Experimental results on large ITC benchmark circuits show that, on the average, the proposed approach reduces run time by 42% and reduces the test size by 46% while having very similar quality as that of timing-aware ATPG.

The second approach generates test patterns for N random long paths simultaneously. *Weighted split vectors* are applied to the algorithm to adjust the probability so that long paths are more likely to be backtraced or propagated than short paths. The weighted split vector is generated by the *static bound analysis*, which calculates static arrival time and propagation time for each gate. Then a novel fault simulation kernel is proposed to quickly calculate the exact path delays of the generated test patterns. The *dynamic bound analysis* is applied to calculate the dynamic propagation time upper bound during logic simulation to avoid redundant fault simulations. A fixed number of test patterns that sensitizes longest paths for each fault are recorded in a compact dictionary, *longest path delay dictionary* (LPDD). LPDD successively reduces memory usage and data communication time between GPU and CPU while preserving the test pattern quality. Finally, a greedy test selection is performed on CPU to select the patterns based on the LPDD. Since only patterns that sensitizes longest paths are selected, the test qualities before and after the test selection are almost the same. Compared with an 8-core CPU-based timing-aware commercial ATPG, our proposed GPU-based technique achieved 36% test length reductions on large benchmark circuits.

To deal with intra-cell defects, *gate exhaustive transition* test generation is proposed . First, the proposed ATPG tries to activate a gate output transition fault by different combinations of gate input transitions. Second, the proposed ATPG tries to propagate fault effects through long paths to increase the chance of detection. Unlike SPICE cell-aware ATPG [Hapke 09], this flow does not require exhaustive SPICE simulation to characterize each library cell. The experiment compared the proposed method with multicore commercial timing-aware ATPG and N-detect transition fault ATPG. Results show that, compared with timing-aware ATPG, test patterns generated by the proposed ATPG improve the test quality by 4%. Also, given approximately the same test length, the proposed ATPG has 4% higher coverage than N-detect transition fault ATPG.

In summary, the proposed framework has shown that it is fast and it is capable of handling multiple complex test metrics.

1.3 Contribution

The proposed framework is the first GPU-based ATPG. This is a three dimensional massively parallel ATPG that cannot be achieved on CPU architecture. Completely different from traditional ATPG, which makes decisions serially and generates single test pattern at a time, the proposed framework generates multiple test patterns targeting multiple test metrics and different faults simultaneously. This framework offers a complete new approach to the current test issues.

The proposed framework is capable of handling multiple test objectives simultaneously and provides N-detect, timing-aware, and cell-aware solutions to deal with complex defect behaviors. N-detect test patterns are generated quickly by targeting multiple faults multiple times simultaneously. Two timing-aware solutions are proposed: 1) fast N-detect test generation and test selection; and 2) targeting multiple long paths simultaneously. The cell-aware solution activates a gate output transition fault by different combinations of gate input transitions and tries to propagate faulty effects through long paths. Compared with multicore CPU-based commercial ATPG, the proposed framework mostly achieved shorter test length and higher test quality in shorter runtime.

1.4 Organization

The organization of this dissertation is as follows. Background on test metrics, ATPG, and GPU are reviewed in Chapter 2. The detail of the SWK algorithm are given in Chapter 3. The detail of GPU Parallelization are given in Chapter 4. The timing-aware and cell-aware test generation are described in Chapter 5. Finally, the conclusion is given in Chapter 6.

Chapter 2

Background

This chapter reviews the past research related to this dissertation. The past work are categorized into three categories: 1) test quality; 2) automatic test pattern generation; and 3) graphical processing unit computation. The test quality section gives reviews of various fault models and test metrics proposed to model defects or measure test quality. The automatic test pattern generation (ATPG) section gives the ATPG development in recent years. The graphical processing unit (GPU) computation section introduces the architecture of GPU.

2.1 Test Quality

Ever since the structural test has become popular in 1970s, fault models and test metrics has continually been an popular research area. The single stuck-at fault model has been the most widely used fault model in both academic and industry [Roth 66]. The single stuck-at fault model assumes only one defect in the chip and the behavior of the defect is stuck-at logic value zero or logic value one. Although there is possibility for stuck-at fault test patterns to detect other untargeted defect, it does not guarantee the detection of all defect. Experiments done in the past already shown that even at 100% stuck-at fault coverage, there are still test escapes [Ma 95]. Many fault models have been proposed ever since to model various kinds of defects. We can categorize the fault models into three categories based on the location or behavior of the defects they are interested in: intra-cell defects, inter-cell defects, and delay defects.

For intra-cell defects, transistor stuck-open and stuck-short have been proposed in [Wadsack 78] [Galiay 80]. For stuck-open faults, two test vectors are required for detection. The first vector sets the activation condition, and the second vector activates the faulty effect. Stuck-short faults, which produces a conducting path between the power and ground, will cause the output voltage to be different as intended. *IDDQ* testing is proposed to measure the steady-state power current to detect stuck-short faults inside the cell [Malaiya 82] [Aitken 99]. Some researchers proposed the *gate exhaustive* (GE)to test all possible input combinations of a cell to detect intra-cell defects [Cho 05]. Take a two-input NAND gate for example, GE ATPG will try to generate all 00, 01, 10, and 11 input combinations. A more recent work proposed modeling of intra-cell bridging defects [Hapke 09]. The authors perform SPICE simulations on every possible defect locations inside a cell and finds out the activation conditions of such defects.

For inter-cell defects, interconnect opens and shorts have been observed in [Friedman 74] [Mei 74] [Roth 76]. Interconnect opens behave like stuck-at faults. Therefore, test patterns with high stuck-at fault coverage is enough to detect interconnect open defects. Interconnect shorts, however, concerns at least two nets. Popular bridging fault models model faulty results as either one net dominated by another net, or there are certain logic relationships between the shorted nets, such as *wired-AND* and *wired-OR*. Recently physical-aware N-detect (PAN) has been proposed as a generallized solution to deal with multiple interconnect bridges [Lin 08]. The idea is to excite as many neighborhood state combinations as possible. For example, support net A has three neighbors, B, C, and D, there are eight possible neighborhood state combinations, namely BCD = {000, 001, 010, . . . , 111}. The ATPG should try to exite all eight state combinations on BCD while targeting stuck-at fault at A. However, generating test patterns for PAN is hard for ATPG due to many test objectives (fault objectives and neighborhood state objectives). Thus, the authors select test patterns effective for PAN from large N-detect test pattern pool.

For delay defects, transition fault model, which assumes a delay defect caused by signal transition at the fault site, has been proposed in [Waicukauski 87]. The transition fault model assumes the extra delay induced by the transition at the fault site is larger than the clock period. Thus, to detect the transition fault, the test pattern must activate the fault-free transition at the fault site and propagate it to the output. The transition fault model has been widely studied and apllied in both academic research and industry. Another popular delay model is path delay fault model [Smith 85]. Path delay fault model considers the cumulative delay along a specific path. In other words, it is the cumulative delay of every transition fault along the path; therefore, it is a more accurate delay model. However, due to the large number of path delay faults in a circuit, it is not practical to generate test patterns for every path delay faults.

With the shrinking geometry and increasing clock frequency, the small delay defects (SDD) is becoming more important over the years. *Statistical delay quality level* was first proposed to measure the quality of test patterns for SDD detection [Sato 05]. Given a delay defect distribution function, *Statistical delay quality level* (SDQL) is calculated to estimate the number of defects not detected by the given test set. The delay defect distribution function is defined as follows.

$$y = F(s) \tag{2.1}$$

where s is the defect size and y represents the probability of a small delay defect of size s. Based on the actual longest path delay slack margin, T_{mgn}, and the sensitized path delay slack margin, T_{det}, the number of undetected sizes for a fault f can be calculated as follows.

$$\int_{T_{mgn}^{f}}^{T_{det}^{f}} F(s_f)ds \tag{2.2}$$

To calculate SDQL, the number of all undetected sizes of all faults are summed.

$$\sum_{f=1}^{|F|} \int_{T_{mgn}^{f}}^{T_{det}^{f}} F(s_f)ds \tag{2.3}$$

The SDQL is an accurate test metric given that the delay defect distribution is accurate. However, the accurate delay defect distribution is hard to obtain since the data is usually confidential. Another delay metric *delay test coverage* (DTC) is proposed in [Lin 06]. DTC evaluates the effectiveness of a test set at detecting transition faults through long paths and is defined as follows.

$$W_f = \frac{PD_f^a}{PD_f^s} \tag{2.4}$$

$$DTC = \frac{\sum\limits_{f \in F} W_f}{|F|} \tag{2.5}$$

where PD_f^s is the *structural longest path* through fault f, and PD_f^a is the *actual longest path* through fault f sensitized by the test set. F is the set of transition faults. Unlike

SDQL, DTC is independent of the clock period and the process-specific delay defect distribution.

2.2 ATPG

Automatic test pattern generator (ATPG) is improving consistently over the past 40 years since the introduction of D Algorithm [Roth 66]. To deal with SDD, ATPG nowadays need to provide timing-aware solutions. Since the amount of delay is small, it is hard to detect using traditional gross delay model such as transition fault. To detect SDD, tests that sensitize fault effects through long paths are needed since the small delay on short paths may be insufficient to cause a faulty response at the output. There are two types of approach: timing-aware ATPG and timing-unaware ATPG.

2.2.1 Timing-Aware ATPG

Due to the large number of paths, generating timing-aware patterns for SDD is very computationally complex. A path-oriented test generator called POTENT based on greedy path expansion is proposed in [Shao 02]. Short paths and intermediate-length paths are masked to force the ATPG to generate patterns through long paths. KLPG selects K longest paths per gate when generating patterns [Qiu 04]. *As late as possible* ATPG has been proposed to detect transition faults via long paths [Gupta 04]. A transition fault ATPG based on SOCRATES with the propagation first or activation first heuristic was proposed in [10]. A hybrid test pattern generation technique, using both timing-aware ATPG and transition fault ATPG, is proposed in [Lee 06]. However, long test length and long runtime make timing-aware ATPG of little practical use for large circuits.

2.2.2 Timing-Unaware ATPG

Instead of timing-aware ATPG, timing-unaware ATPG has been applied with some modification [Peng 10]. Short paths and intermediate paths are masks to force ATPG to generate patterns along long paths. After test generation, a subset of patterns is then selected. Alternatively, selecting patterns from a large N-detect test set is also effective [Goel 09].

Since exact calculation of sensitized path length is slow, output deviation has been used as an alternative measure to select patterns [Yilmaz 08a]. Layout information can be also considered to take crosstalk effect into account [Yilmaz 08b]. Although output deviation calculation is very fast, it is just an indirect metric for path length.

2.2.3 Parallel ATPG

Parallel programming is a popular technique to speed up ATPG [Klenke 92]. Parallel ATPG algorithms on CPU can be divided into three categories according to different partitioning schemes: fault partitioning [Cai 10], search space partitioning [Patil 90], and circuit partitioning [Smith 87]. The fault partitioning approach dynamically or statically partitions the fault list into multiple partitions and each of which is handled by a different core. The search space partitioning approach partitions the solution space into different portions, each of which is searched by different cores. The circuit partitioning approach divides the circuit into multiple subcircuits, each of which is taken care of by a core. Recent research showed that two to five times of speedup has been achieved using fault partitioning approach on eight-core CPU [Cai 10].

2.3 GPU Computation

GPU has become a highly parallel general computing device which supports up to tens of thousands of threads running concurrently. Our ATPG is implemented using *compute unified device architecture* (CUDA) on NVIDIA's graphic card. A GPU *device* contains hundreds of *stream processors*. A task running on GPU is a *kernel*, which specifies how each thread is executed. To manage the threads running on GPUs, CUDA provides a three-level architecture: *grid*, *block*, and *thread*. On GPU, all threads of the same tasks reside in the same grid. Each grid consists of multiple blocks and each block consists of multiple threads. The core executing the thread can access the corresponding required data by its grid ID, block ID, and thread ID. In the particular GPU we use in our experiment, GeForce GTX590, the maximum allowable blocks within a grid is 248 and the maximum allowable threads within a block is 210. Though theoretically 258 of threads can be invoked at the same time, the actual number of threads is limited by the memory capacity of the GPU.

The memory architecture of the GPU plays a crucial role in the program performance. Table 2.1 shows the type, level, latency, and size of different available memories on our NVIDIA GTX590 GPU.

Table 2.1: GPU Memory Specification

Type	Level	Latency cycles (cached)	Size	Accessibility
Global	Grid	400-600	1.5GB	R/W
Texture	Grid	400-600 (4-6)	1.5GB	Read-only
Constant	Grid	400-600 (4-6)	64KB	Read-only
Shared	Block	4-6	48KB	R/W
Register	Thread	immediate	32KB	R/W

The memory architecture can be divided into three levels: grid-level, block-level, and thread-level. Grid-level memories, *global*, *constant*, and *texture*, can be accessed by all threads. Constant and texture memories are read-only with cache whereas global memory is readable and writable without cache. If cache hit, the latency can be reduced approximately from 600 cycles to 6 cycles. Global and texture memory reside on the DRAM of the GPU and they share same memory space. The user can specify a block of memory to be texture memory. Constant memory has only 64KB and is often used for storing constant coefficient. *Shared memory* is block-level memory space where every thread within the block can access. The shared memory size is 48KB per block and its latency is 4 to 6 cycles. Thread-level *registers* are only seen by each thread. The latency of register is very short but its size is limited to 32KB per block.

Chapter 3

SWK ATPG Algorithm

This chapter presents the detail of the core algorithm of the proposed ATPG framework, the *Split-into-W-Clones* (SWK) algorithm. SWK is a word-level parallel algorithm that is capable of targeting multiple test objectives simultaneously. As mentioned in Chapter 1, we should not rely on single test metric to achieve maximum effectiveness, so SWK provides a solution to set different objectives to a group of clones. Unlike tranditional ATPG which makes decisions serially, SWK converts decision making into parallel bitwise logic operation so multiple decisions can be searched at the same time. In SWK, when a two-way decision is needed, the SWK algorithm randomly splits into two groups of *clones*, each of which performs its own independent search. The quality of SWK test patterns is higher than that of traditional ones because the former achieve multiple objectives at the same time. Also, SWK test patterns contains don't care bits so they can be easily compressed by on-chip DFT hardware.

3.1 SWK Concept

The core concept of the SWK algorithm is generating multiple test patterns through word-level parallelization. To illustrate this concept, a simple stuck-at fault test generation example is given here to point out the difference between the proposed algorithm and the traditional ATPG. The following subsections first describes the signal encoding and notation used in this work followed by an example to show the SWK concept.

3.1.1 SWK signal encoding

In SWK, signal A is represented by eight words of W bits: A^0, A^1, A^d, $A^{\bar{d}}$, A^x, A^{b_0}, A^{b_1} and A^p. Each bit in a word represents an individual *clone*, which performs an independent search in the decision tree. The meaning of the first five words is as follows. $A_k^0 = 1$: signal A is *0* (good 0 / faulty 0) for the k_{th} clone. $A_k^1 = 1$: signal A is *1* (good 1 / faulty 1) for the k_{th} clone. $A_k^d = 1$: signal A is d (good 1 / faulty 0) for the k_{th} clone. $A_k^{\bar{d}} = 1$: signal A is $\bar{d}$ (good 0 / faulty 1) for the k_{th} clone. $A_k^x = 1$: signal A is x (unknown) for the k_{th} clone. For a given clone, the above five values are mutually exclusive so at most one of them equals one at a time. When A is unknown, the other three words indicate whether A is on one of two paths: the *propagation path* and the *objective path*. The former is a path that faulty effect (d or $\bar{d}$) will potentially propagate to reach an output. The latter is a path that the objective backtrace follows to reach an input. For the k_{th} clone, $A_k^p = 1$: signal A is on a propagation path. $A_k^{b_0} = 1$: signal A is on an objective backtrace path (objective value = 0). $A_k^{b_1} = 1$: signal A is on an objective backtrace path (objective value = 1). Again, these three values are mutually exclusive so at most one of them equals one at a time. Table 3.1 shows an example of signal encoding using 8-bit words, which means there are eight clones. The last four rows of the table show the logic value of each clone (bit), whether the clone is on the propagation path, whether the clone is objective backtrace path (Y for yes and N for no), and the notation used in figures to represent the value of the clone, respectively. From the table, it can be observed that the first four clones have logic values of *0*, *1*, d, $\bar{d}$, respectively, and the last four clones have logic values of unknown. Also, The last three clones are on propagation path, objective backtrace 0 path, and objective backtrace 1 path, respectively. Clones on propagation path are denoted as p and clones on objective backtrace path are denoted as b_0 or b_1.

Table 3.1: Signal encoding example

Word	Clones							
	1	2	3	4	5	6	7	8
A_k^0	1	0	0	0	0	0	0	0
A_k^1	0	1	0	0	0	0	0	0
A_k^d	0	0	1	0	0	0	0	0
$A_k^{\bar{d}}$	0	0	0	1	0	0	0	0
A_k^x	0	0	0	0	1	1	1	1
A_k^p	0	0	0	0	0	1	0	0
$A_k^{b_0}$	0	0	0	0	0	0	1	0
$A_k^{b_1}$	0	0	0	0	0	0	0	1
Logic value	*0*	*1*	d	$\bar{d}$	x	x	x	x
Propagation	N	N	N	N	N	Y	N	N
Objective	N	N	N	N	N	N	Y	Y
Notation	*0*	*1*	d	$\bar{d}$	x	p	b_0	b_1

3.1.2 Difference between SWK and traditional ATPG

Fig. 3.1 and Fig. 3.2 show how tradtional ATPG (such as PODEM or FAN) generates a test pattern for single stuck-at faults. The example circuit has four inputs (A, B, C, H) and two outputs (Q, R). Suppose that the target fault is line G stuck-at zero. Fig. 3.1 shows the input assignments after the backtrace, ACH=*100*. Fig. 3.2 shows the circuit values after the implication. From the figure, we can see that the d-frontier disappears after the implication as both $G1$ and $G2$ are masked by the side inputs. In this case, a backtrack is needed to reassign inputs. In this example, the tranditional ATPG made a decision on whether to assign A = *1* or B = *1* to achieve the primary objective. If the wrong decision (A = *1*) is made, a backtrack is required to remake the decision, which is costly in time.

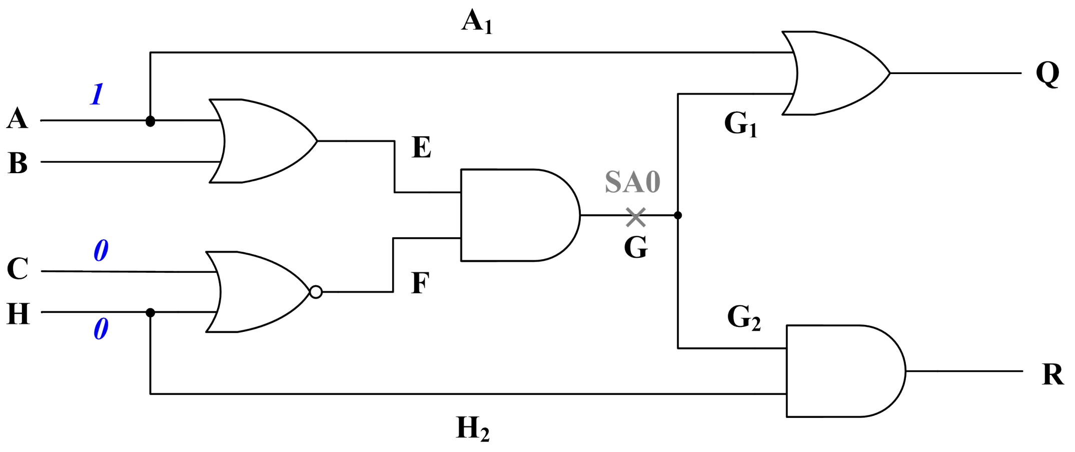

Figure 3.1: Tranditional ATPG example (backtrace).

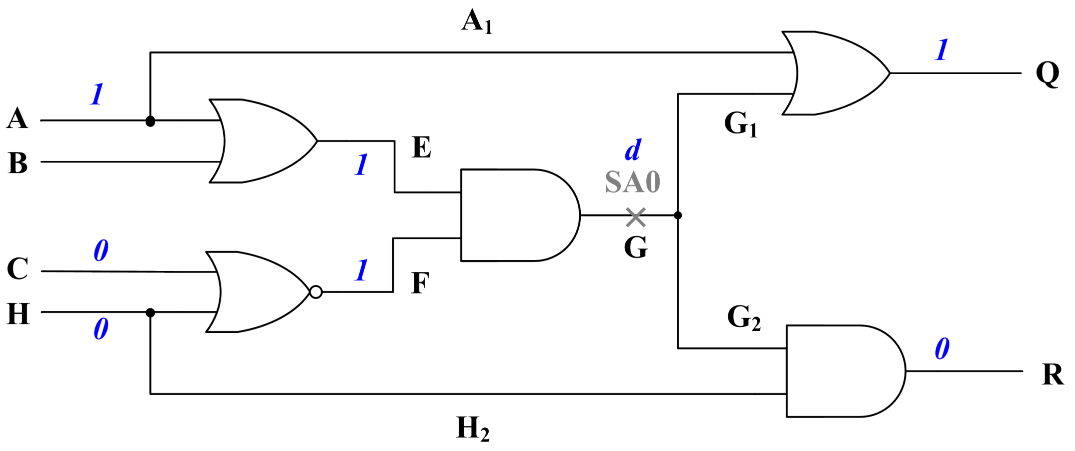

Figure 3.2: Tranditional ATPG example (implication).

For the same circuit and same target stuck-at fault, Fig. 3.3 through Fig. 3.5 illustrates how SWK algorithm generates test pattern. Fig. 3.3 shows the first backtrace of the SWK algorithm. In this example, two-bit word are used ($W = 2$) to represent two clones: clone #1 and clone #2. The initial objectives for both clones are both ones, which is $\{b_1, b_1\}$ in the figure. Backtracing gate G is an implication backtrace because both gate inputs must be one to justify the output objective b_1. Both E and F are assigned the same objective values for both clones. Backtracing gate E requires a decision because either $A = 1$ or $B = 1$ justifies the output objective b_1. The SWK algorithm performs an *objective split* (*o-split*), by which two clones are assigned different objectives. In this example, clone #1 backtraces b_1 on input A whereas clone #2 backtraces b_1 on input B. This is denoted as $A = \{b_1, x\}$ and $B = \{x, b_1\}$ in the figure. Both inputs C and H are assigned zeros because gate F is implication backtrace.

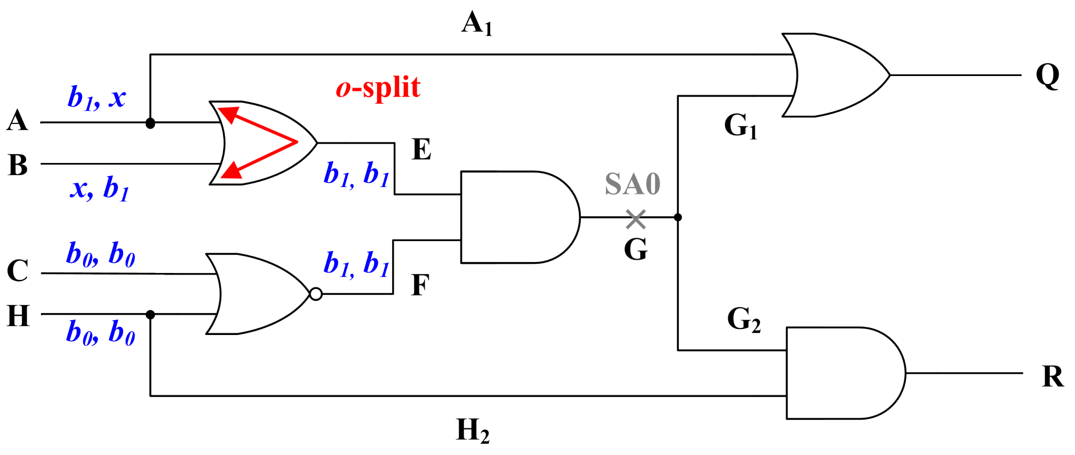

Figure 3.3: SWK example (1_{st} backtrace).

Fig. 3.4 shows the propagation after Fig. 3.3. The fault is excited for both clones so line G is now $\{d, d\}$ in the figure. Output Q is one for clone #1 but unknown for clone #2, denoted as $\{1, p\}$ in the figure. The symbol p indicates that output Q is on the propagation path. Output R is $\{0, 0\}$ for both clones. Because no propagation path reaches an output, clone #1 is now dead. There is one propagation path remains so clone #2 is still alive.

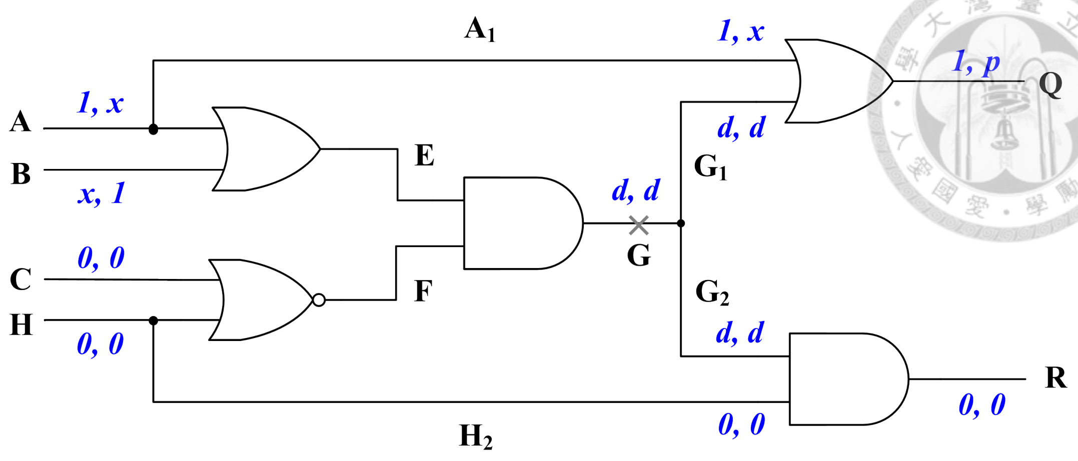

Figure 3.4: SWK example (1_{st} propagation).

Fig. 3.5 shows the second backtrace followed by another propagation. To propagate the error, input A is backtraced a zero. The clone #2 successfully propagates a d to output Q. A test pattern ($ABCH$=0100) is successfully generated by clone #2. Totally, SWK needs two parallel backtraces and two parallel propagations without any backtrack.

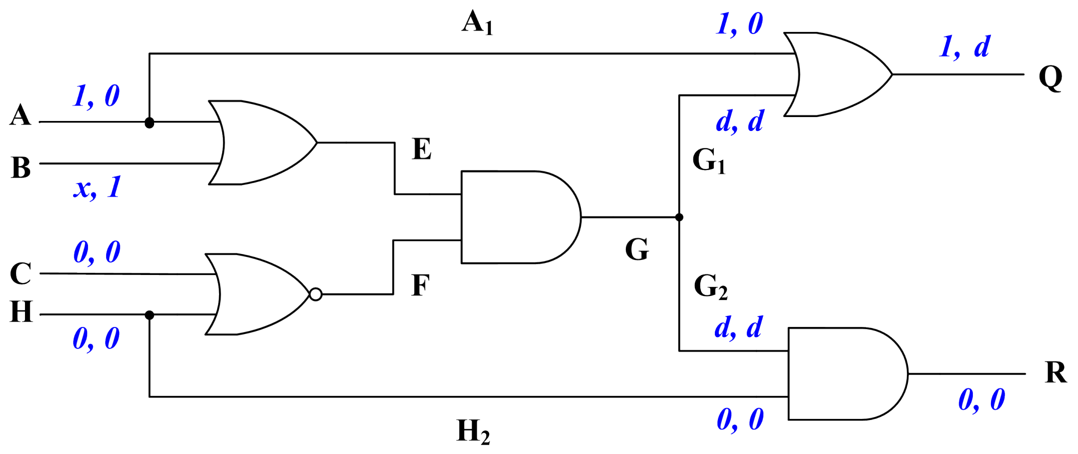

Figure 3.5: SWK example (2_{nd} backtrace and then propagate).

3.2 SWK Algorithm

Algorithm 1 shows the overall SWK algorithm. In the preprocess stage, the netlist is parsed and levelized. Given a fault list, an undetect fault is first selected as target fault and *TestGeneration* is performed to generate a packet of W test patterns. If the result is

SUCCESS or PARTIAL_SUCCESS, the *DynamicTestCompaction* function is then executed to compact more undetected faults into t. Whether x-filling is needed or not depends on the application of hardware compactor. After the x bits are filled, fault simulation is performed to drop the detected faults. Patterns that detect undetected faults is added to the test set. The generation continues until all faults have been tried or time out. Finally, the *StaticTestCompaction* function is performed to remove those patterns that do not improve the quality of any metric. The generated test set T is returned after fault grading.

Algorithm 1 SWK ATPG

```
 1: procedure SwkAtpg($F$)
 2:     preprocess;
 3:     $T \leftarrow$ empty test set
 4:     repeat
 5:         $f \leftarrow$ choose undetected fault from $F$
 6:         $t \leftarrow$ initialized inputs of $W$ test patterns
 7:         $result \leftarrow$ TestGeneration($f$, $t$)
 8:         if $result$ = SUCCESS or PARTIAL_SUCCESS then
 9:             DynamicTestCompaction($t$)
10:         end if
11:         $x$-filling
12:         fault dropping
13:         $T \leftarrow T \cup t$ that detect undetected faults
14:     until time out or no more undetected faults
15:     StaticTestCompaction($T$)
16:     fault grading
17:     return $T$
18: end procedure
```

Algorithm 2 shows the pseudo code of the test generation. For stuck-at test generation, the initial objective simply controls the faulty net to the opposite value of the stuck value. A backtrace is then performed from the fault site to the inputs. After inputs are assigned, a propagation is performed from inputs to the outputs. A clone is successful if a d or $\bar{d}$ has reached the output. Backtrack is performed on the dead clones. A clone is dead if either 1) it fails to achieve any one of its objectives; or 2) it achieves objectives but there is no propagation from f to an output. The SWK algorithm returns SUCCESS as soon as the number of successful clones is equal to or larger than the specified N. When a specified runtime is up, the algorithm aborts with two outcomes: some clones are successful (PARTIAL_SUCCESS), or no clone is successful (FAIL).

Algorithm 2 Test Generation

```
 1: procedure TestGeneration($f$, $t$)
 2:     InitializeObjective($f$)
 3:     repeat
 4:         Backtrace()
 5:         AssginInputs($t$)
 6:         Propagate()
 7:         if # successful clones $\geq N$ then
 8:             return SUCCESS
 9:         end if
10:         Backtrack($t$)
11:     until time out
12:     if # successful clones > 0 then
13:         return PARTIAL_SUCCESS
14:     else
15:         return FAIL
16:     end if
17: end procedure
```

3.2.1 Propagation

In SWK, a zero delay event driven parallel pattern logic simulator is needed (see Algorithm 3). The propagation is done by evaluating the logic value of each gate or fanout level by level from inputs to outputs. Evaluation is only needed if any inputs of the gate or fanout have changed. The propagation is very similar to traditional logic simulator except for four additional operations: *d-generation*, *d-propagation*, *p-generation*, and *p-propagation*.

Algorithm 3 Propagation

```
1: procedure propagate
2:     $l \leftarrow 0$
3:     repeat
4:         for gate or fanout $g$ in logic level $l$ do
5:             if any input of $g$ changed then
6:                 evaluate($g$)
7:                 $d$-generation and $d$-propagation
8:                 $p$-generation and $p$-propagation
9:             end if
10:        end for
11:        $l \leftarrow l + 1$
12:    until $l >$ maximum logic level
13: end procedure
```

A d or $\bar{d}$ is generated when the fault site is controlled to the opposite value of the stuck value. d-generation can be easily performed by inserting a logic operation at the fault site. For A stuck-at fault, the corresponding d-generation can be simply implemented by the following equations.

$$A^d = A^1, \text{ for } A \text{ stuck-at zero fault} \tag{3.1}$$

$$A^{\bar{d}} = A^0, \text{ for } A \text{ stuck-at one fault} \tag{3.2}$$

After d-generation, the faulty effect (d or $\bar{d}$) is propagated from a gate input to its gate output when 1) at least one of the input is d or $\bar{d}$; and 2) none of the gate input holds the controlling value. For the example of a two-input AND gate, with inputs AB and output C, the following logic operations implement d-propagation.

$$C^1 = A^1B^1 \tag{3.3}$$

$$C^0 = A^0 + B^0 + A^dB^{\bar{d}} + A^{\bar{d}}B^d \tag{3.4}$$

$$C^d = A^1B^d + A^dB^1 + A^dB^d \tag{3.5}$$

$$C^{\bar{d}} = A^1B^{\bar{d}} + A^{\bar{d}}B^1 + A^{\bar{d}}B^{\bar{d}} \tag{3.6}$$

where multiplication represents the bitwise AND logic operation and addition represents the bitwise OR logic operation. For an inverter with input A and output C, the equations are simply as follows.

$$C^1 = A^0 \tag{3.7}$$

$$C^0 = A^1 \tag{3.8}$$

$$C^d = A^{\bar{d}} \tag{3.9}$$

$$C^{\bar{d}} = A^d \tag{3.10}$$

Since we have shown the equations of both AND gates and inverters, the equations of any other gates can be derived in the same way. The values of fanout branches are the same as those of the fanout stem.

In propagation, a $A^p = 1$ represents that net A is on a possible propagation path of the fault effect. A p is generated when one of the gate inputs is d or $\bar{d}$ while the other input is unknown. A p could be propagated from gate input to gate output when three conditions are met: 1) one of the input is p; 2) the other input is non-controlling value; and 3) the gate output is unknown. For a two-input AND gate, the *p-generation* and *p-propagation* can be written as the following bitwise logic operation.

$$C^p = \underbrace{A^x(B^d + B^{\bar{d}} + B^x(A^d + A^{\bar{d}})}_{p\text{-generation}} + \underbrace{\overline{A^0}B^pC^x + A^p\overline{B^0}C^x}_{p\text{-propagation}} \tag{3.11}$$

where first two terms represent p-generation and the last two terms representa p-propagation. For an inverter, p cannot be generated; it can only be propagated.

$$C^p = A^p \tag{3.12}$$

Similarly, for a fanout stem, p cannot be generated but only propagated. For a fanout stem M with n fanout branches M_1, M_2, $\ldots M_n$, p is propagated if the fanout stem is p.

$$M_1^p = M^p, M_2^p = M^p, \ldots, M_n^p = M^p \tag{3.13}$$

Fig. 3.6 illustrates a larger example of propagation. Suppose that B stuck-at zero is our target fault. The figure shows the circuit after the propagation. After B is assigned to $\{1, 1\}$, faulty effects $\{d, d\}$ are generated at the fanout branches of B. Since the side inputs of the two d-frontiers, G_1 and G_2, are unknowns, $\{p, p\}$ is generated at the output of the d-frontier gates. After p-generation, p is propagated through G_3 and G_4 to output Q. In this example, after propagation two paths are marked as possible propagation paths for the faulty effect: $G_1 \to G_4 \to Q$ and $G_2 \to G_3 \to G_4 \to Q$.

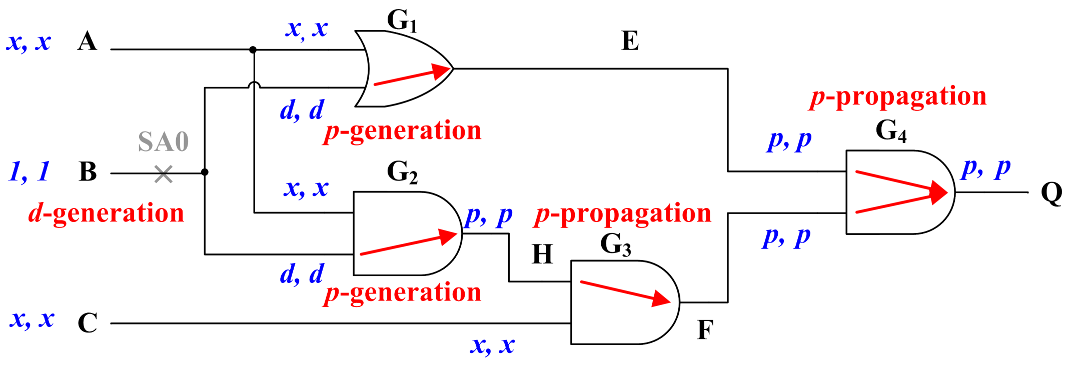

Figure 3.6: SWK propagation example.

3.2.2 Backtrace

After the propagation is finished, a backtrace (see Algorithm 4) is performed level by level from outputs to inputs. The backtrace function can be divided into two phases: the *p-backtrace* and the *o-backtrace*. The former searches for path(s) to propagate the fault effect and the latter searches for input assignment(s) to achieve the objective. The OBJ flag indicates whether an objective has been generated or not. Please note that every clone is independent so each clone can be in different backtrace phase at the same time.

Algorithm 4 Backtrace

```
1: procedure backtrace
2:     $l \leftarrow$ maximum logic level
3:     $OBJ \leftarrow 0$
4:     repeat
5:         for gate or fanout $g$ in logic level $l$ do
6:             $p$-backtrace($g$)
7:             $o$-backtrace($g$, $OBJ$)
8:             $OJB \leftarrow$ generated objectives
9:         end for
10:        $l \leftarrow l - 1$
11:    until $l < 0$
12: end procedure
```

After the fault is activated, *p-backtrace* is needed to choose a path to propagate fault effects. During the p-backtrace, $A^p = 1$ represents that net A is on a candidate propagation path. There are two types of p-backtraces: *p-implication* and *p-split*. A p-implication is performed when there is only one possible propagation path in p-backtrace. Fig. 3.7 shows an example of p-implication. Before the backtrace, the value of the upper input is $\{p, p\}$ and the value of the lower input is $\{1, 1\}$. Thus, the p values at the output can only be backtraced to the upper input. Another scenario is that the two gate inputs before backtrace are both $\{p, p\}$. In this case, a p-split is performed to split the p's to different paths. Fig. 3.8

shows the example of p-split, where one p backtraces the upper gate input while the other p backtraces the lower gate input.

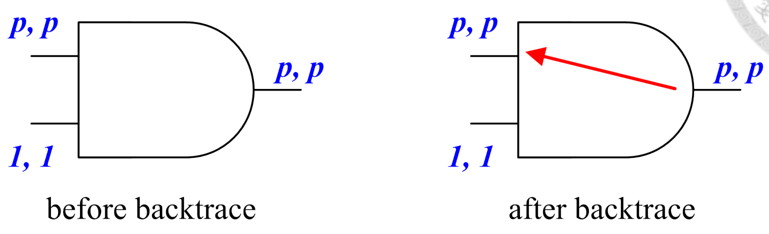

Figure 3.7: Example of p-implication.

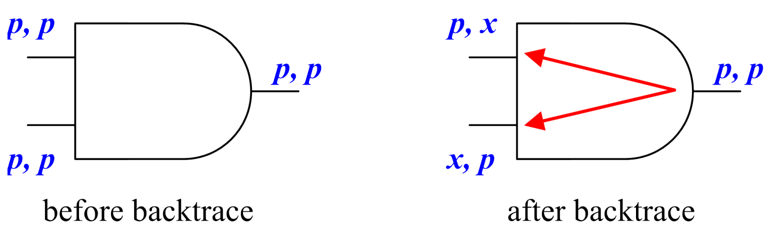

Figure 3.8: Example of p-split.

The following equations implement the p-backtrace of an AND gate with inputs AB and output C.

$$A^p = \underbrace{A^p(B^1 + \overline{B^p})C^p \cdot \overline{OBJ}}_{p\text{-implication}} + \underbrace{A^pB^pC^pS \cdot \overline{OBJ}}_{p\text{-split}} \tag{3.14}$$

$$B^p = \underbrace{B^p(A^1 + \overline{A^p})C^p \cdot \overline{OBJ}}_{p\text{-implication}} + \underbrace{A^pB^pC^p\overline{S} \cdot \overline{OBJ}}_{p\text{-split}} \tag{3.15}$$

where the first term is simply an p-implication and the second term corresponds to a p-split. Please note that the p's on the right hand side are obtained during the propagation and the p's on the left hand side are obtained during the backtrace.

S is a random *split vector* of W bits, half of which are ones and the other half are zeros. Sample split vectors and their complements are listed below.

$$S_1 = [000\ldots0111\ldots1] \qquad \overline{S_1} = [111\ldots1000\ldots0]$$

$S_2 = [1010101010\ldots10]$ $\overline{S_2} = [0101010101\ldots01]$

$S_3 = [11001100\ldots1100]$ $\overline{S_3} = [00110011\ldots0011]$

where S_i and $\overline{S_i}$ is a pair of complement split vectors. In our implementation, whenever a p-split is needed, one pair of split vectors is randomly chosen such that all objective split are uncorrelated.

As for an inverter, p simply backtraces from the output to the input; no p-split is needed.

$$A^p = C^p \cdot \overline{OBJ} \tag{3.16}$$

When backtracing a fanout stem, the fanout stem is p as long as one of the fanout branches is p; otherwise, the fanout stem is not p. For a fanout stem M with n fanout branches M_1, M_2, ...M_n, backtrace can be implemented in the following equations.

$$M^p = (\sum_{i=1}^{n} M_i^p) \cdot \overline{OBJ} \tag{3.17}$$

where the summation represents a serial of n-1 bitwise OR logic operations.

An objective can be either a b_1 (backtrace one) or a b_0 (backtrace zero). In stuck-at fault test generation, each clone has at most one objective at a time. The status of the objective generation is store in a W-bit OBJ flag vector. $OBJ_k = 1$ means that the k_{th} clone has an objective generated; otherwise, the k_{th} clone is still waiting for an objective to be generated. The OBJ vector is updated every time the propagation is finished. Before the fault is excited, the initial objective is to control the faulty line to its opposite value of the stuck value. If line A stuck-at one is our target fault, then the initial objective generation is shown in the two following equations. The former generates the objective value; the latter sets the OBJ flag to prevent another objective from being generated again.

$$A_0^b = A^x \tag{3.18}$$

$$OBJ = A^x \tag{3.19}$$

After the fault is excited, the objective is to propagate d or $\bar{d}$ to an output. An objective is generated at the gate input when 1) no objective has been generated so far; and 2) the gate is a d-frontier. For the same example of a two-input AND gate with inputs AB and output C, the two following equation handles *o-generation*. The former generates an objective when the gate is a d-frontier (B = d or $\bar{d}$, and A = x) and there is no existing objectives. The latter sets the control flag to prevent another objective from being generated again for the same clone. Fig.3.9 shows the o-generation corresponding to equations.

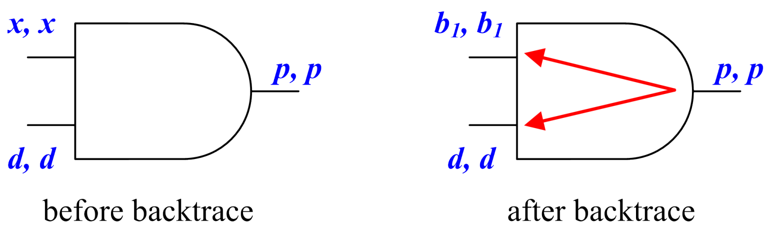

Figure 3.9: Example of o-generation.

There are two types of o-backtraces: *o-implication* and *o-split*. An o-implication is unique whereas an o-split has two choices. Fig. 3.10 and Fig. 3.11 shows two examples of o-implication for an AND gate. In Fig. 3.10, the lower gate input is one (non-controlling) so b_0 can only backtrace to the upper gate input. In Fig. 3.11, both gate inputs are unknowns so b_1 backtraces to both gate inputs. Fig. 3.12 shows an example of o-split for an AND gate. Both gate inputs are unknown so b_0 is split into two: one b_0 backtraces the upper gate input while the other b_0 backtraces the lower gate input.

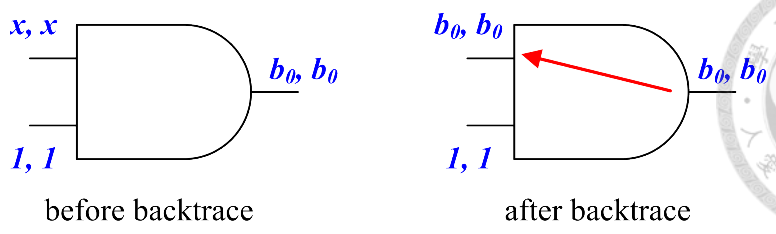

Figure 3.10: Example of o-implication on one gate input.

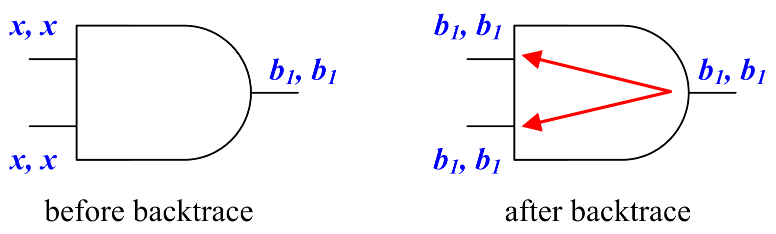

Figure 3.11: Example of o-implication on two gate inputs.

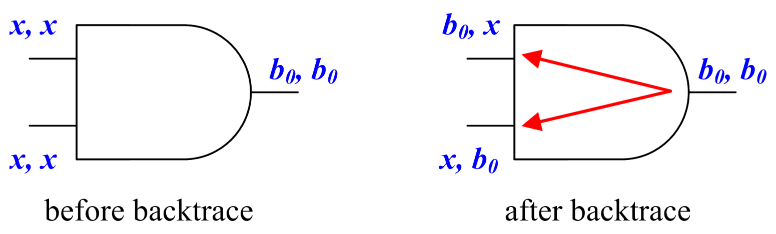

Figure 3.12: Example of o-split.

Putting together o-generation, o-implication, and o-split, backtracing a two-input AND gate with inputs AB and output C can be written as follows.

$$A^{b_1} = \underbrace{A^x(B^d + B^{\bar{d}})C^p \cdot \overline{OBJ}}_{o\text{-generation}} + \underbrace{A^x(B^0 + B^{\bar{d}})' \cdot C^{b_1}}_{o\text{-implication}} \tag{3.20}$$

$$A^{b_0} = \underbrace{A^x(B^1 + B^d + B^{\bar{d}})C^{b_0}}_{o\text{-implication}} + \underbrace{A^p B^p C^{b_0} S}_{o\text{-split}} \tag{3.21}$$

$$B^{b_1} = \underbrace{B^x(A^d + A^{\bar{d}})C^p \cdot \overline{OBJ}}_{o\text{-generation}} + \underbrace{B^x(A^0 + A^{\bar{d}})' \cdot C^{b_1}}_{o\text{-implication}} \tag{3.22}$$

$$B^{b_0} = \underbrace{B^x(A^1 + A^d + A^{\bar{d}})C^{b_0}}_{o\text{-implication}} + \underbrace{B^p A^p C^{b_0} \overline{S}}_{o\text{-split}} \tag{3.23}$$

where a prime sign indicates a complement. The equations for input B are the same as those for input A except that the split vector in the former is the complement of that in the latter. This is to make sure that the clones are split into two groups without overlap.

When backtracing a fanout stem, the rule is to backtrace the value that all branches agree upon. The fanout stem equals to b_1 if at least one fanout branch equals to b_1, and none of fanout branches equals to b_0. The fanout stem equals to b_0 if at least one fanout branch equals to b_0, and none of fanout branches equal to b_1. The fanout stem performs a random objective split when both b_0 and b_1 appear in fanout branches. For a fanout stem M with n fanout branches M_1, M_2, $\ldots M_n$, backtrace can be implemented in the following bitwise logic operation.

$$M^{b_0} = (\sum_{i=1}^{n} M_i^{b_0})(\overline{\sum_{i=1}^{n} M_i^{b_1}}) + S \cdot (\sum_{i=1}^{n} M_i^{b_0})(\sum_{i=1}^{n} M_i^{b_1}) \tag{3.24}$$

$$M^{b_1} = (\sum_{i=1}^{n} M_i^{b_1})(\overline{\sum_{i=1}^{n} M_i^{b_0}}) + \overline{S} \cdot (\sum_{i=1}^{n} M_i^{b_0})(\sum_{i=1}^{n} M_i^{b_1}) \tag{3.25}$$

Fig. 3.13 is an follow up backtrace example of Fig. 3.6. Backtrace starts from output Q towards inputs ABC. When backtracing gate G_4, since both gate inputs are unknowns, a p-split is performed. The result is clone #1 backtraces the upper gate input whereas clone #2 backtraces the lower gate input. Gate G_3 performs p-implication since only the upper gate input has the propagation value. o-generations are perform at the d-frontiers and backtrace objectives are generated at the inputs of G_1 and G_2. The two different objectives b_0 and b_1 are merged at the fanout branch of A. In this example, SWK simultaneously backtraced two different objectives $\{b_0, b_1\}$ at A.

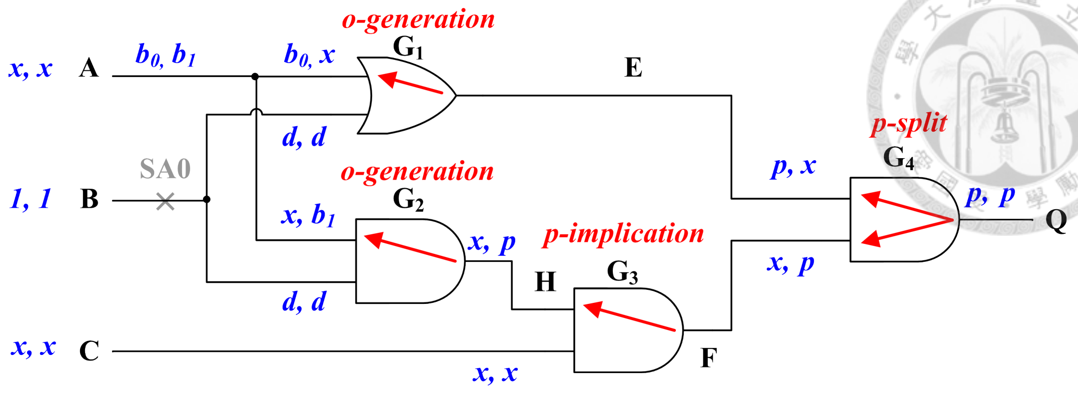

Figure 3.13: SWK backtrace example.

3.2.3 Assign Inputs

Inputs are assigned to its backtrace objective values after each backtrace. After assignment, each clone will record its own input assignment value and order in a *decision stack* for possible backtracks. Algorithm 5 shows the pseudo code of input assignment.

Algorithm 5 Assign Inputs

```
1: procedure AssignInputs($t$)
2:     for each input $pi$ do
3:         $pi \leftarrow$ backtrace objective value
4:     end for
5:     each clone update decision stack
6:     $t \leftarrow$ all input values
7: end procedure
```

Fig. 3.14 is a follow up example to Fig. 3.13 to show the input assignment. In this example, A is set to $\{b_0, b_1\}$ since its backtrace objective value is $\{b_0, b_1\}$. After the input assignment, when faulty effects have not reached outputs, the test generation continues to perform backtrace, input assignment, and propagation until test succeeds or time out. Fig. 3.15 shows the result of finished test generation of the same example. In this case, after another two backtraces and propagations, the second clone successfully generated one test pattern ABC = $\{111\}$. The first clone, on the other hand, is dead and requires backtrack.

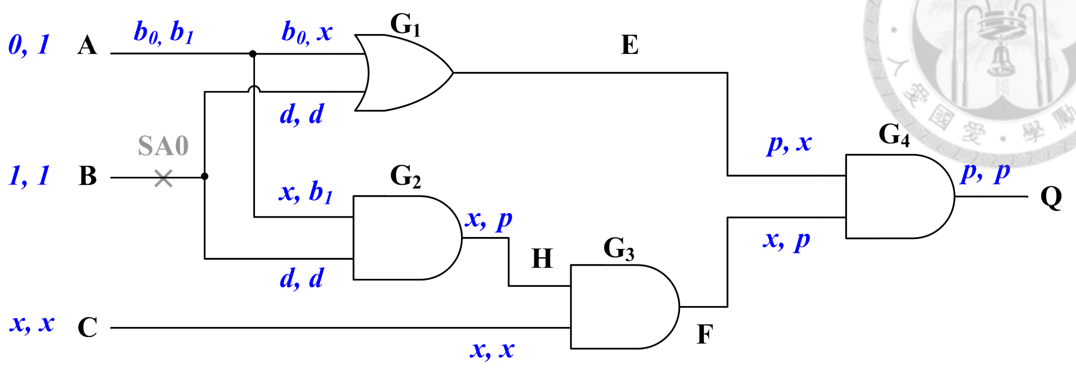

Figure 3.14: SWK assign inputs example.

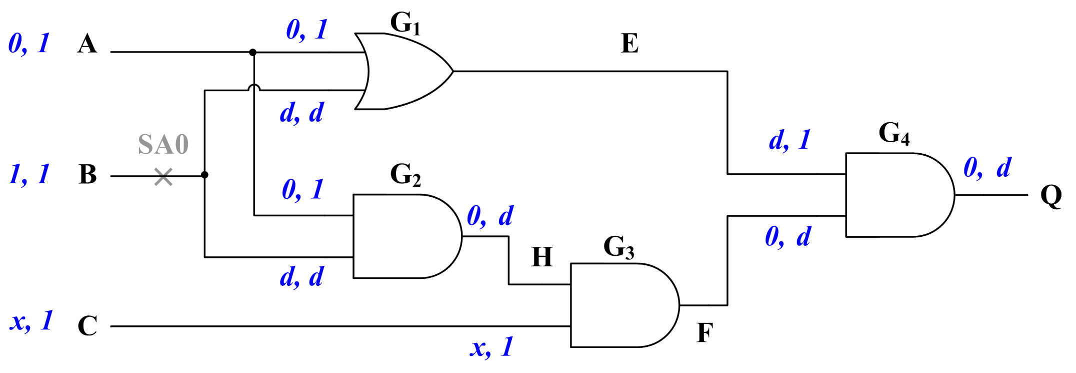

Figure 3.15: SWK finished test generation.

3.2.4 Backtrack

Every time a propagation is finished, SWK backtracks the dead clone(s), if any. A clone is *dead* if it has no chance of propagting the faulty effect to any of the outputs. Whick means, if all outputs are not p, d, or $\bar{d}$, the clone is dead. Every clone has a unique decision stack which stores the input that has been assigned. When backtrack occurs, the last input is popped out of the stack. The value of the input would be flipped if the complement of the input has not been tried; otherwise, both values of the input have been tried and we just change it to unknown and the second to last input is backtracked. Algorithm 6 shows the pseudo code of the backtrack process. The backtrack mechanism of SWK is similar to that of PODEM except that multiple inputs are assigned at once. Even though multiple inputs are assigned during one backtrace, each clone keeps a unique ordering of the assigned inputs and will start backtrack from the last assignment. Given enough computation time, SWK returns a solution if there exist one. In other words, SWK is a complete algorithm.

Algorithm 6 Backtrack

```
procedure Backtrack($t$)
    check output to find dead clones
    for all clones that are dead do
        $stack$ ← the decision stack
        $assignment$ ← pop from $stack$
        while $assignment$ has been flipped do
            if $stack$ is empty then
                return UNTESTABLE
            end if
            $assignment$ ← pop from $stack$
        end while
        $t$ ← flip $assignment$
        push $assignment$ into $stack$
    end for
end procedure
```

Fig. 3.16 and Fig. 3.17 illustrate the backtrack process continued from the previous example in Fig. 3.15. In Fig. 3.16 it can be observed that output clone #1 is dead since the value of output Q is $\{0, d\}$. The decision stack of clone #1 is shown in the top of figure. In this case, the assignment $A = 0$ will be popped and flipped to *1*. The result of the backtrack is shown in Fig. 3.17. From the figure, we can see that after backtrack, clone #1 generated the same test pattern as clone #2, which is the only solution in this example. In summary, the backtrack process of SWK is similar to those of traditional ATPG except SWK has keeps multiple decision stacks, where each clone stores its decision stack and performs backtrack independently.

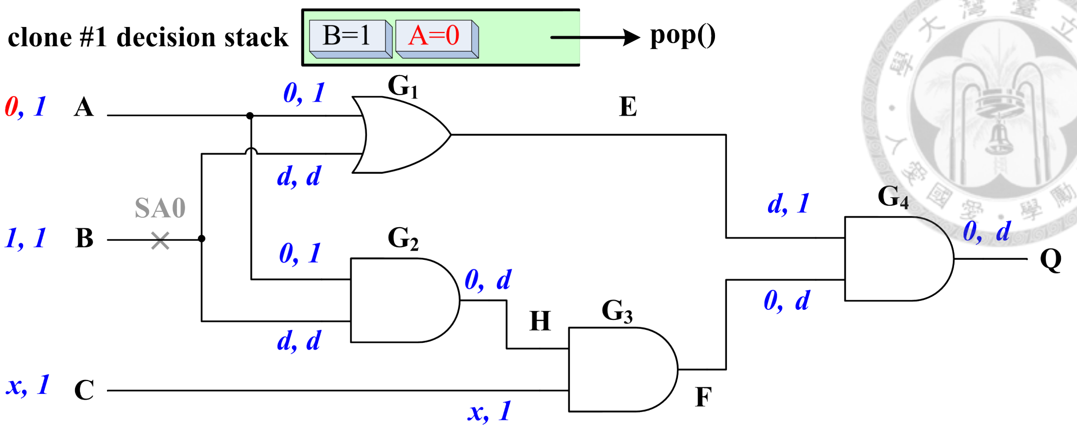

Figure 3.16: SWK find dead clones.

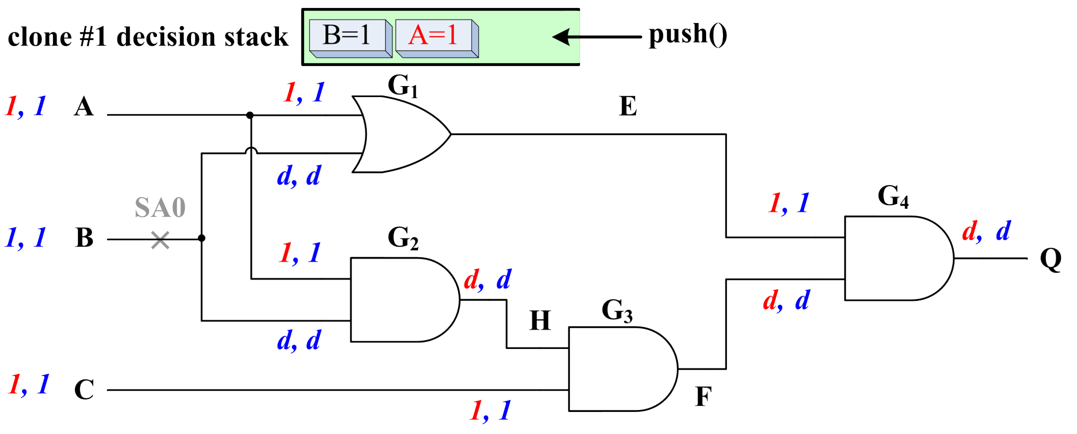

Figure 3.17: SWK backtrack dead clones.

3.2.5 Initialize Multiple Objectives

An important feature of SWK is its capability to deal with multiple objectives. When generating stuck-at fault test patterns, SWK has only one objective —the fault objective. However, when generating better test patterns, SWK has to achieve other objectives — *quality objectives*. The fault objective is unique and it has to be met; otherwise the clone is dead. The quality objectives, on the other hand, may be missed.

We demonstrate the multiple objectives test generation by targeting stuck-ar fault, N-detect, PAN, and BCE. Algorithm 7 shows how SWK initialize these objectives. The fault objective simply controls the fault site to the opposite value to the faulty value. For N-

detect, PAN, and BCE coverages, SWK assigns different quality objectives based on the neighborhood states. Before test generation, a list of neighbors for each net is extracted from the physical information. Suppose that the number of neighbors of the fault site is m so the number of distinct neighborhood states is 2^m. If 2^m is less than W, then $W/2^m$ objectives are evenly assigned to each clone; if 2^m is greater than W, then only the first W objectives are assigned to each clone.

For GE coverage, SWK assigns different objectives to the faulty gate input. If the excitation of target fault requires o-implication, e.g. stuck-at zero fault of an AND gate output, then all W clones are assigned the same objective. If the excitation of target fault requires o-split, e.g. stuck-at one fault of an AND gate output, then three combinations are evenly distributed among W clones.

Please note SWK is not limited to only the mentioned fault or quality objectives. Any objectives that can be transformed into backtrace objective values can be supported by SWK. For example, a slow-to-rise transition fault can be transformed into b_0 in the first time frame and b_1 in the second time frame at the fault site. Another example is path delay fault. SWK can set the side inputs of the targeted paths to non-controlling backtrace objectives.

Algorithm 7 Initialize Objectives

```
procedure InitializeObjective($f$)
    $n \leftarrow$ faulty net of $f$
    if $f$ is stuck-at one then
        $n^{b_0} \leftarrow 1$
    else
        $n^{b_1} \leftarrow 1$
    end if
    $m \leftarrow$ number of neighbors of $n$
    if $2^m \leq W$ then
        $Q \leftarrow$ all possible neightborhood states
    else
        $Q \leftarrow W$ different neightborhood states
    end if
    assign $Q$ quality objectives to $W$ clones
    if excite $f$ requires $o$-split then
        $C \leftarrow$ all possible gate input combinations
    end if
    assign $C$ quality objectives to $W$ clones
end procedure
```

3.2.6 Dynamic Test Compaction

Fig. 3.18 shows an example of SWK dynamic test compaction. For the primary fault, G stuck-at one fault, two test cubes has been generated: $\{00xx\}$ generated by clone #1 and $\{xx11\}$ by clone #2. The corresponding outputs are $\{\bar{d}, x\}$ and $\{x, \bar{d}\}$, respectively. Suppose that we choose Q stuck-at zero fault as the secondary fault. Although clone #1 is dead for this secondary fault, clone #2 is still alive. Clone #2 backtraces a b_1 from Q to A. Assigning A to one, followed by a propagation, SWK successfully detects the secondary fault without backtrack (Fig. 3.19).

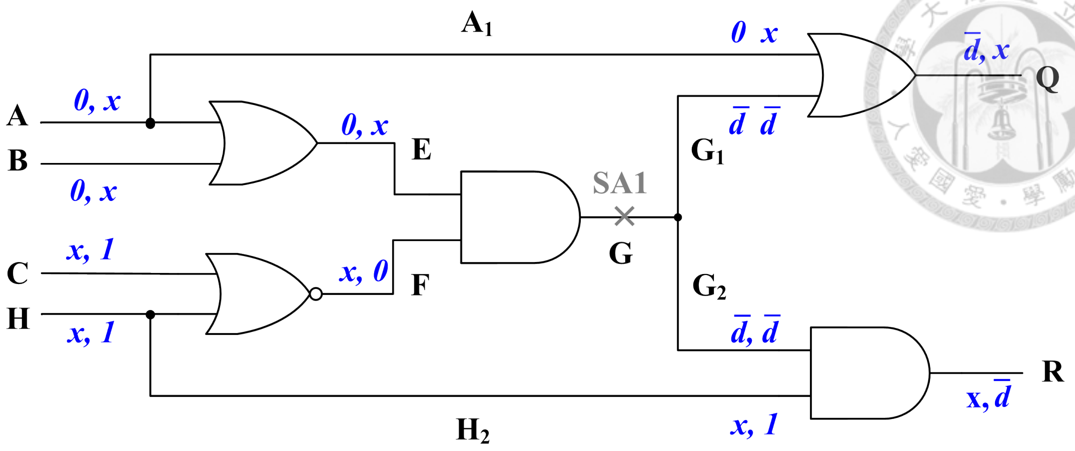

Figure 3.18: SWK dynamic test compaction example (before compaction).

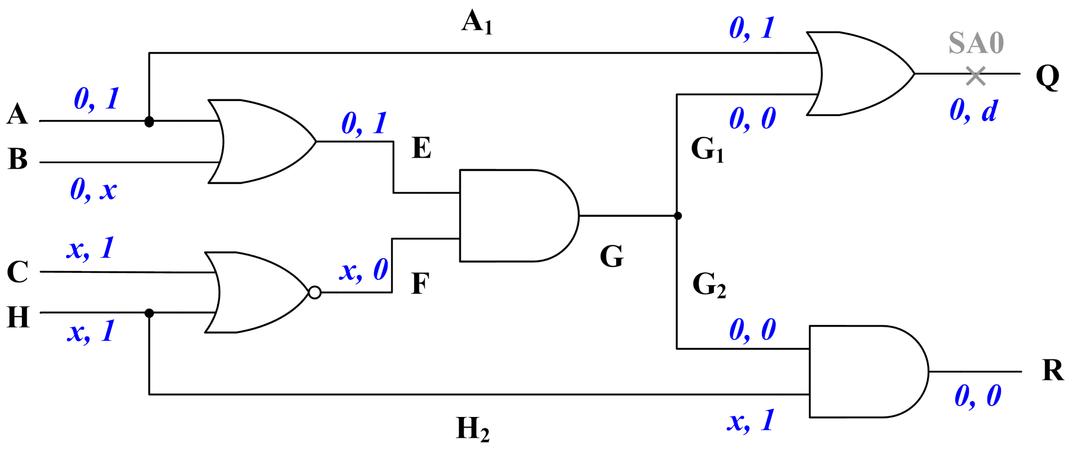

Figure 3.19: SWK dynamic test compaction example (after compaction).

The dynamic test compaction algorithm is shown in Algorithm 8. A secondary fault is picked from a net n in the fanin cone of an output with many x's. The algorithm picks a net n that has an undetected fault and many unspecified values x in many clones. Test generation for the secondary fault is the same as that of the primary target fault. The dynamic test compaction repeats until time out or the abort count is larger than a predefined limit.

Algorithm 8 Dynamic Test Compaction

```
1: procedure DynamicTestCompaction($t$)
2:     $abortCount \leftarrow 0$
3:     repeat
4:         $n \leftarrow$ a net in fanin cones of an output with many $x$
5:         if $n$ has undetect fault $f$ and many $x$ then
6:             if TextGeneration($f$, $t$) is FAIL then
7:                 $abortCount \leftarrow abortCount + 1$
8:             end if
9:         end if
10:    until time out or $abortCount$ larger than limit
11: end procedure
```

3.2.7 Static Test Compaction

Those test cubes that do not contribute to any quality metric (N-detect, GE, BCE, PAN coverages) are removed in static test compaction. Static test compaction is simply implemented by pattern reordering and fault simulation. For GE and PAN coverage, the number of distinct states is stored for each fault during fault simulation. To speed up this process, a parallel pattern single fault propagation simulator is implemented to simulate W patterns at a time.

3.2.8 Complex Cells

Advanced designs often contain complex gates such as multi-input cells or tri-state buffers. Gates with three or more inputs has backtrace and propagation equations similar to those of the two-input gates except for the split vector used. Take three-input AND gate with inputs ABC and output Y for example. The following equations implement the propagation operation.

$$Y^p = A^x(B^d + B^{\bar{d}})(C^d + C^{\bar{d}}) + B^x(A^d + A^{\bar{d}})(C^d + C^{\bar{d}}) + C^x(A^d + A^{\bar{d}})(B^d + B^{\bar{d}})$$
$$(A^0 + B^0)'C^pY^x + (B^0 + C^0)'A^pY^x + (A^0 + C^0)'B^pY^x \tag{3.26}$$

The backtrace equations for the three-input AND gate are listed below.

$$A^{b_1} = A^x(B^d + B^{\bar{d}} + C^d + C^{\bar{d}})Y^p \cdot \overline{OBJ} + A^x((B^0 + B^{\bar{d}})(C^0 + B^{\bar{d}}))'Y^{b_0} \tag{3.27}$$

$$A^{b_0} = A^x(B^1 + B^d + B^{\bar{d}})(C^1 + C^d + C^{\bar{d}})Y^{b_0} + (A^p(B^p + C^p)')Y^p \cdot \overline{OBJ} \tag{3.28}$$

$$A^p = A^p(A^1 + (B^p + C^p)')Y^p \cdot \overline{OBJ} + A^pB^pC^pY^pS \cdot \overline{OBJ} \tag{3.29}$$

Backtracing equations for inputs B and C are similar to input A. Split vector for three-input gates is different from those for two-input gates. The number of ones in the split vector for three-input gates is 1/3 of the CPU word size. Below is a sample split vector.

$$S = [\underbrace{111\ldots1}_{1/3}\underbrace{000\ldots0}_{2/3}]$$

Although the number of bitwise operations of three-input AND gate is higher than those of two-input AND gate, it is still lower than the sum of operations of splitting the three-input into two two-input AND gates.

To handle tri-state logic, we need new encoding vector to represent high-Z logic. A tri-state signal A need two additional words of W bits: A^z and A^{b_z}. A^z denotes whether A is high-Z. A^{b_z} denotes whether A is on objective high-Z backtrace path. Note that the extra encoding vector z is used only at tri-state logic to save memory. Given a tri-state buffer with input A, enable signal E, and output C, the following equations implement the propagation operation.

$$C^1 = E^1A^1 \tag{3.30}$$

$$C^0 = E^1A^0 \tag{3.31}$$

$$C^d = E^1A^d \tag{3.32}$$

$$C^{\bar{d}} = E^1A^{\bar{d}} \tag{3.33}$$

$$C^z = E^0 \tag{3.34}$$

$$C^p = \overline{E^0}A^p + E^d + +E^{\bar{d}} + \overline{E^p}(A^d + A^{\bar{d}}) + E^p \tag{3.35}$$

The backtrace equations for the tri-state buffer are listed below.

$$A^{b_1} = A^x((E^x + E^1)C^{b_1} \cdot OBJ + E^dC^pS \cdot \overline{OBJ}) \tag{3.36}$$

$$A^{b_0} = A^x((E^x + E^1)C^{b_0} \cdot OBJ + E^dC^p\overline{S} \cdot \overline{OBJ}) \tag{3.37}$$

$$\begin{aligned} E^{b_1} &= E^x[(A^x + A^1 + A^d)C^{b_1} + A^xC^{b_0}] \\ &\quad + E^x[(A^0 + A^{\bar{d}})C^{b_0} + (A^d + A^{\bar{d}})C^p \cdot \overline{OBJ}] \end{aligned} \tag{3.38}$$

$$A^p = A^x(E^xC^pS + \overline{E^x}C^p) \cdot \overline{OBJ} \tag{3.39}$$

$$E^{b_0} = E^xA^xC^{b_z} \cdot \overline{OBJ} \tag{3.40}$$

$$E^p = E^x(A^xC^p\overline{S} + A^x\overline{C^p}) \cdot \overline{OBJ} \tag{3.41}$$

3.3 Experimental Results

A multiple objective experiment is performed to show SWK , stuck-at fault N-detect, PAN, BCE, and GE are used for grading. PAN coverage is defined as $S_d(f)$ over $S_p(n, f)$, where $S_p(n, f)$ is the minimum of n and the number of possible neighborhood states. For example, if fault f_1 has 3 neighbors, then $S_p(n = 5, f_1) = min(5, 2^3) = 5$. But if fault f_2 has only 2 neighbors, then $S_p(n = 5, f_2) = min(5, 2^2) = 4$. Similarly, $S_d(f)$ is the minimum of n and the number of distinct neighborhood states when fault f is detected. Please note that the number n used for the fault grading, which is five, is different from the number N used for pattern generation.

$$PANCoverage(n) = \sum_{\forall f} (\frac{S_d(f)}{S_p(n, f)}) \times 100\% \tag{3.42}$$

Table 3.2 and Table 3.3 list the experimental results of SWK test sets for large ISCAS'89 benchmark circuits and two IWLS'05 benchmark circuits. The number in the parenthesis is gate count of each circuit. All benchmark circuits are synthesized, placed, and routed by commercial tools. The bridging pairs are extracted from the layout DEF files of each benchmark. The extracted bridging pairs are used in test generation and PAN coverage calculation. The numbers of traditional N-detect test sets, generated by a commercial ATPG tool, are also shown for reference. In both ATPG, test cubes with x are generated first, followed by random x-filling, which simulate a typical DfT and ATPG flow with an on-chip test pattern compressor.

Table 3.2 shows the runtime (RT) and test length (TL) comparison between SWK and commercial ATPG. The first row for each circuit shows SWK test sets (N = 5) and the traditional N-detect test sets of the same N. The second line for each circuit shows very long N-detect test sets, where the N is set to 50 (or the highest feasible value) in the commercial ATPG. There are two runtimes for SWK. The first one is the run time used to generate 5-detect stuck-at fault patterns. The second one is the run time used to generate patterns targeting all four quality metrics.

Table 3.3 shows the coverage of the four test metrics: N-detect, PAN, BCE, and GE.

Table 3.2: SWK runtime and test length comparison

Circuit	Commercial ATPG			SWK ATPG			
	N	RT(s)	TL	N	N-det RT(s)	4-metric RT(s)	TL
s13207 (8K)	5	1.8	1,502	5	6.3	20.4	12,990
	50	12.5	29,419				
s15850 (10K)	5	1.6	877	5	6.1	19.0	9,357
	15	10.7	31,045				
s35932 (18K)	5	2.7	107	5	15.7	43.8	1,540
	50	14.4	1,868				
s38417 (24K)	5	3.2	2,282	5	65.4	237.4	9,377
	50	20.9	29,190				
s38584 (21K)	5	4.5	707	5	52.7	188.4	7,311
	40	26.2	11,475				
aes_core (26K)	5	6.1	1,134	5	68.6	258.7	6,746
	50	40.5	9,817				
des_perf (105K)	5	16.3	1,279	5	154.1	524.6	3,334
	50	159.1	12,655				

It can be observed that, the BCE coverage of both test sets are approximately the same. GE coverage of SWK test sets is about 1% to 2% higher than those of traditional N-detect test sets. PAN coverage of the former is 10% to 20% higher than that of the latter. The experimental data show that SWK test sets are better than the test sets selected from very large N-detect test sets.

Fig. 3.20 compares the PAN and GE coverage of commercial N-detect ATPG and SWK generated patterns under different test length on benchmark s38584. The eleven nodes of the commercial N-detect ATPG represent 1-detect to 10-detect and 20-detect pattern results. The four nodes of SWK represent 1-detect to 4-detect patterns. From the

Table 3.3: SWK fault coverage comparison

Circuit	Commercial ATPG					SWK ATPG			
	N	N-det	PAN	BCE	GE	N-det	PAN	BCE	GE
s13207	5	99.13	77.74	97.73	94.90	98.21	89.52	98.49	94.92
(8K)	50	99.13	81.60	98.47	94.88				
s15850	5	98.08	67.72	96.64	92.83	98.04	88.23	97.08	93.03
(10K)	15	98.07	70.05	97.10	92.78				
s35932	5	91.06	61.09	89.42	81.86	90.96	81.10	89.64	81.95
(18K)	50	91.06	61.44	89.49	81.85				
s38417	5	99.40	79.60	98.68	96.65	99.10	89.81	99.20	97.05
(24K)	50	99.40	81.89	99.25	96.60				
s38584	5	95.63	72.73	94.78	91.19	95.48	86.35	95.10	92.07
(21K)	50	95.63	74.71	95.09	91.25				
aes_core	5	99.98	64.57	96.31	92.57	99.97	75.48	97.22	92.98
(26K)	50	99.98	67.83	97.14	92.57				
des_perf	5	100.00	63.75	98.85	88.24	100.00	75.64	99.04	89.82
(105K)	50	100.00	68.14	99.12	89.76				

figure we can draw two conclusions: 1) patterns generated by SWK have better PAN and GE coverage than those generated by commercial N-detect ATPG given the same test length and 2) patterns selected from very large N-detect pattern pool cannot achieve same or higher PAN or GE coverage than patterns generated by SWK.

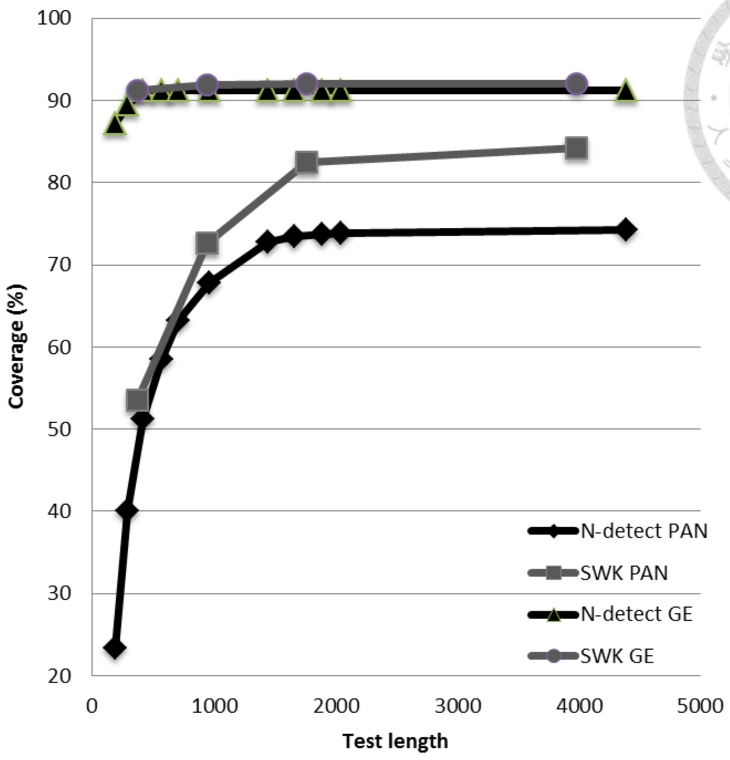

Figure 3.20: Coverage as a function of test length.

3.4 Parallelism Analysis

The performance of SWK is analyzed by measuring the runtime using different word size W and different number of detection N. The word size is equal to the number of clones performing bitwise logic operations at the same time. The number of detection determines how many clones target the same fault at a time. For example, if $W = 4$ and $N = 8$, the first eight clones set their initial objectives to the target fault but only 4 clones can perform logic operations at the same time.

Table 3.4 shows the runtime comparison between SWK and an in-house FAN algorithm based ATPG. From the table we can see that, as W increases, runtime decreases. The runtime of $W = 64$ and $W = 128$ are roughly the same is because we are using a 64-bit machine to run this experiment. Any W that is greater than 64 has to be processed serially. Take $W = 1$ and $N = 1$ for example, the runtime is 22.3 times longer than FAN. This number is quite reasonable since the number of backtrace bitwise logic operations for a gate is between 20 to 30 depending on the type of the gate. SWK really shines when it comes to large N-detect test generation. Suppose $W = 64$ and $N = 4$, SWK is 4.27 times faster than FAN. If N is increased to 64, SWK is 4.44 times faster than FAN. This shows that SWK scales well with the number of detection.

Table 3.4: Runtime analysis using different W and N (s38584)

W	N				
	1	4	8	32	64
1	475.4	123.9	2,603.3	8,914.0	16,284.7
4	120.3	426.4	743.8	2,448.1	4,513.5
8	61.2	193.8	413.2	1,410.2	2,582.1
32	15.9	45.1	96.1	402.9	679.3
64	8.3	18.3	42.9	169.3	387.5
128	8.2	18.1	41.8	165.8	380.4
FAN	21.3	78.1	206.5	895.4	1,715.7

Another test generation experiment for hard-to-detect faults is performed to show that SWK word-level search space parallelism helps to reduce test generation time for hard-to-detect faults. The benchmark circuit b19 is used for the experiment. Stuck-at faults that are aborted by commercial ATPG using abort limit equals to 3 are considered as hard-to-detect faults. In this circuit, there are a total of 3,350 hard-to-detect faults. Different abort limits, 5, 10, 50, and 100, are used for commercial tool to generate test patterns for these hard-to-detect faults. SWK uses different N and different abort limits to generate test patterns for the same set of faults. Table **??** shows the number of detected faults, number of untestable faults, number of abort faults, and the runtime of the ATPGs under different abort limit and different N settings. From the table, we can see that using $N = 4$ and abort limit = 10, SWK successfully generated test patterns for all testable faults while the runtime is shorter than the commercial ATPG.

Table 3.5: Search space parallelism analysis on hard-to-detect faults (b19)

	abort limit	N	#detected	#untestable	#abort	runtime
Commercial	5	1	2,852	498	135	9.4
	10	1	2,953	519	13	107.8
	50	1	2,963	521	1	414.7
	100	1	2,963	522	0	431.5
SWK	5	1	2,841	493	151	17.6
		2	2,945	503	37	22.7
		4	2,963	521	1	52.9
		8	2,963	521	1	99.3
	10	1	2,962	520	3	37.2
		2	2,963	521	1	59.4
		4	2,963	522	0	101.9

3.5 Summary

This chapter presents the SWK parallel ATPG algorithm to generate test patterns that meet multiple objectives. SWK uses random split to convert decisions into parallel bitwise logic operations so that multiple objectives can be tried at the same time. Four test metrics, N-detect, PAN, BCE, and GE are targeted simultaneously by SWK in the experiments. Results show that PAN coverage of SWK test patterns is 10% to 20% higher than that of high quality 50-detect ATPG patterns while the former is shorter.

Chapter 4

GPU Parallelization

A fast GPU-based is proposed to accelerate SWK algorithm. All three components of ATPG system are parallelized on GPU: test generation, logic simulation, and fault simulation. In test generation, three levels of parallelism are implemented: *device*-level, *block*-level and *word*-level partitioning. The partitioning techniques in each level are as follows: 1) device-level fault partitioning; 2) block-level fault and circuit partitioning; and 3) word-level fault and search space partitioning.

Device-level fault partitioning statically partitions faults into different fault lists, each of which is assigned to a different device. Block-level fault partitioning assigns different target faults to different blocks, whereas block-level circuit partitioning assigns different logic levels to different blocks. World-level fault partitioning assigns target faults to different bits in a word, whereas word-level search space partitioning assigns different branches of the decision tree to different bits. That means, the proposed algorithm converts decision making into bitwise logic operation so different branches of the decision tree can be explored at the same time. Suppose the number of devices is v, the number of blocks is t, and the size of a word is W. Our ATPG generates patterns concurrently.

For example, if v = 2, t = 64, and W = 32, then 4,096 patterns can be generated simultaneously. This is a massively parallel ATPG that cannot be achieved on traditional CPU architecture.

4.1 GPU Porting

Fig. 4.1 shows the overall flow of our ATPG system. In the preprocess stage, the netlist is parsed, levelized and then duplicated into two time frames. After the preprocess stage, the amount of memory needed for storing netlist and fault list is allocated on the GPU. The *test generation kernel* is then invoked by CPU to generate test patterns for undetected faults. After test generation kernel, the *fault simulation kernel* is invoked to simulate the test patterns. The number of detection of each fault is recorded. A fault is dropped if the number of detection is larger than or equal to N. The generated test patterns are zero-copied to CPU main memory after each simulation. At the same time, test generation continues to generate tests for remaining faults. After all faults have been tried, test grading is performed to determine the quality of the generated test set.

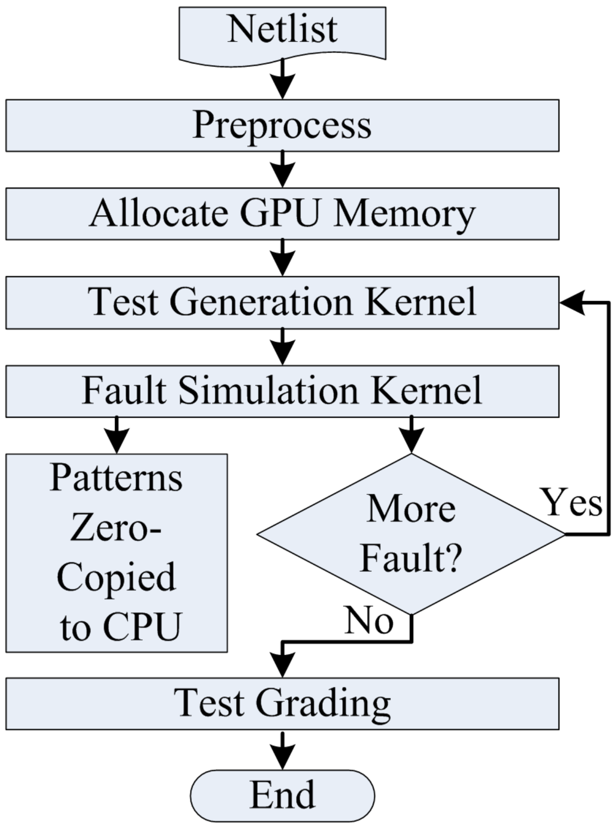

Figure 4.1: GPU ATPG flow.

4.1.1 GPU Memory Allocation

Fig. 4.2 shows memory allocation of our ATPG on GPU. Netlist is allocated in the grid-level texture memory since it is read-only and accessed by all threads. For each gate in the netlist (such as $G0$), information such as gate type and fanin/fanout are stored in a slot. Because 128 gates in a logic level are accessed simultaneously, empty slots are filled with dummies. Fault list is allocated in the global memory to store the location and the number of detection of each fault. Circuit signals are stored in global memory. Each of t blocks keeps its own circuit signals so that block-level parallel test generation can access these signals independently. Temporary values during run time of test generation and fault simulation are stored in the shared memory of each block.

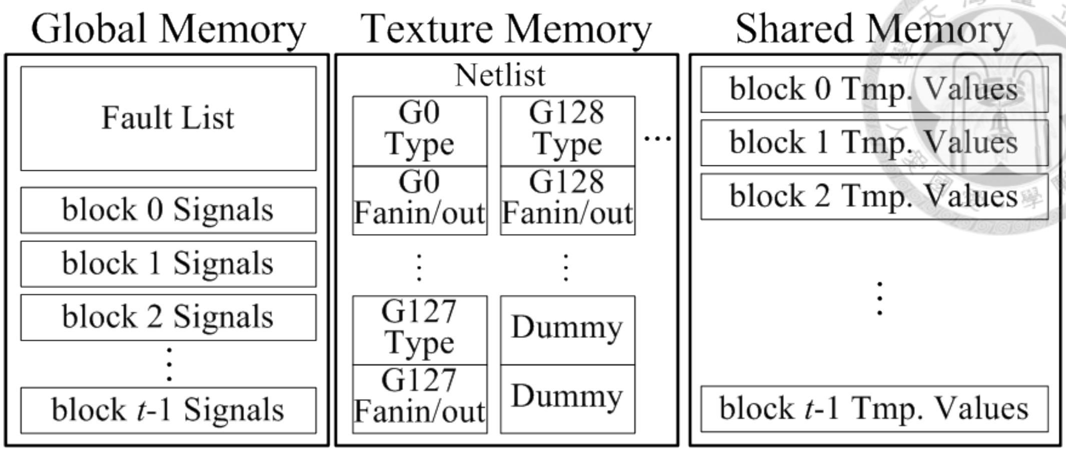

Figure 4.2: Memory Allocation on GPU.

4.1.2 GPU Memory Access Optimization

Memory access plays a crucial role in the performance of GPU programs. Three techniques have been implemented to optimize global memory access latency, access efficiency, and communication with CPU main memory.

1) The netlist information, such as gate types and connections, is allocated in texture memory to reduce global memory access latency. Texture memory has its own cache which can reduce access latency by one hundred times if data is cached.

2) Coalesced access is a NVIDIA GPU feature to maximize global memory access efficiency. Coalesced access requires data access to be aligned to 128-byte cache lines —that means data addresses must start at multiples of 128. To utilize coalesced access, signal values are packed into 128-bytes and aligned to cache lines.

3) Zero-copy technique is implemented to transfer generated patterns back to main memory to hide the communication overhead. A block of main memory is allocated as zero-copy memory. After each fault simulation, the patterns are “zero-copied” back to main memory while GPU continues to generate patterns for undetected faults at the same time.

4.2 Test Generation Kernel

The test generation kernel consists of three levels of parallelism: *word*-level parallelism for a *block*, *block*-level parallelism for a kernel, and *device*-level parallelism for a circuit under test (CUT). Each block executes a word-level parallel SWK test generation algorithm and each device handles a statically partitioned fault list. The following subsections describe each level of parallelism in detail with a simple example.

4.2.1 Word-Level Parallel Test Generation for a Block

The word-level parallel test generation implements the SWK algorithm to handle N-detect transition faults test generation on GPU. Figure 3 shows the test generation flow of each block in our GPU-based test generation for N-detect transition delay faults. Given a set of faults F, word size W, and the target number of detection N, each fault in F is handled by N clones. For example, if W = 32, F = f_1, f_2, f_3, f_4, and N = 8, then the first eight clones generate patterns for f_1, and the following eight clones generate patterns for f_2, and so on. For transition faults test generation, the two initial objectives are fault-free values at the fault site in two time frames. A backtrace is then performed from the fault site to the inputs. After inputs are assigned, fault effect propagation is performed from inputs to the outputs. The test generation of a clone is successful if a d or has reached any output. This process ends if test generations for all W clones are successful or time limit has reached.

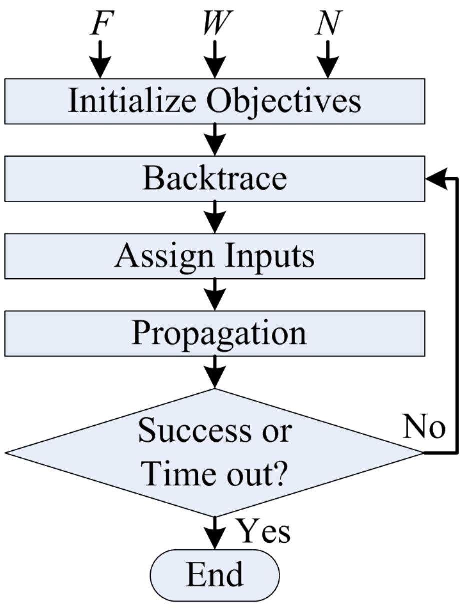

Figure 4.3: Test generation kernal flow for a single block.

Fig. 4.4 to Fig. 4.7 illustrate an example to generate a test pattern for a single transition fault. This is a two-time frame circuit, each of which consists of three AND gates, three inputs (E, F, G), and one flip-flop (H). LV stands for logic level. Suppose that the target fault is $G2$ slow-to-fall. We use a two-bit word (W=2) to represent two clones: clone #1 and clone #2. Since the fault is slow-to-fall, the initial objectives in time frame one and two are one and zero, respectively. They are denoted as $\{b_1, b_1\}$ and $\{b_0, b_0\}$ in Fig. 4.4.

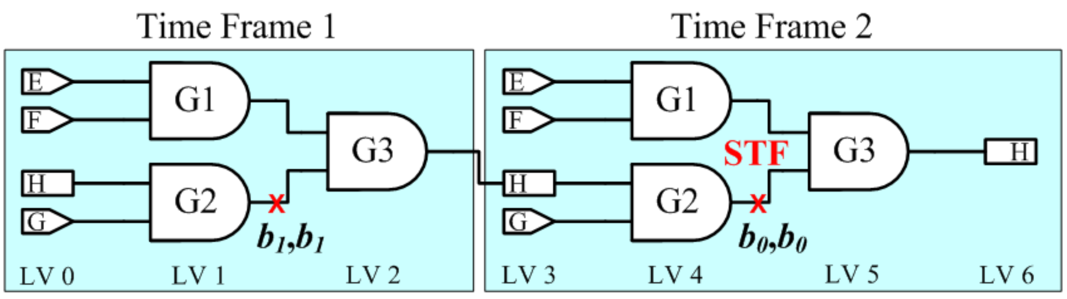

Figure 4.4: Test generation example (initialize objectives).

Fig. 4.5 shows the first backtrace after Fig. 4.4. Backtracing $G2$ in time frame one is an *implication backtrace* because both gate inputs must be one to justify the output objective b_1. Backtracing $G2$ in time frame two requires a decision because either H = 0 or G =

0 justifies the output objective b_0. An *objective split* is performed to assign two clones with different objectives. In this example, clone #1 backtraces b_0 on input H and clone #2 backtraces b_0 on input G.

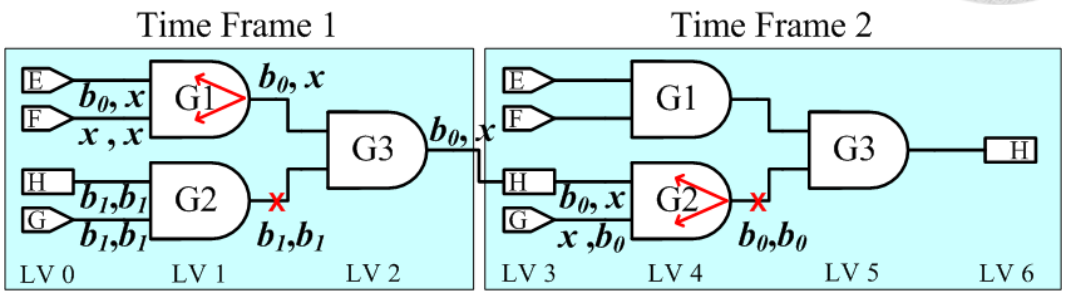

Figure 4.5: Test generation example (first backtrace).

Fig. 4.6 shows the propagation after Fig. 4.5. The fault is excited for both clones so $G2$ in time frame two is $\{\bar{d}, \bar{d}\}$ in the figure. Output H is $\{p, p\}$ in the figure.

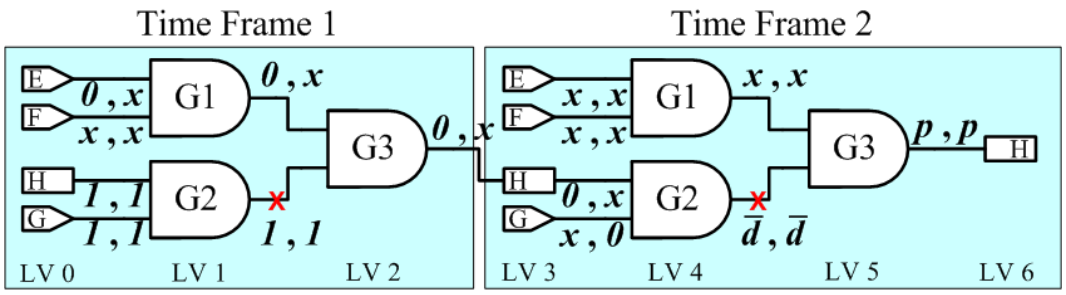

Figure 4.6: Test generation example (first propagation).

Fig. 4.7 shows the second backtrace followed by another propagation. To propagate the faulty effect to output, both clones of E and F in time frame two are set to one. Two test patterns EFG = {(*0x1*, *11x*), (*xx1*, *110*)} and H = {*1*, *1*} are successfully generated.

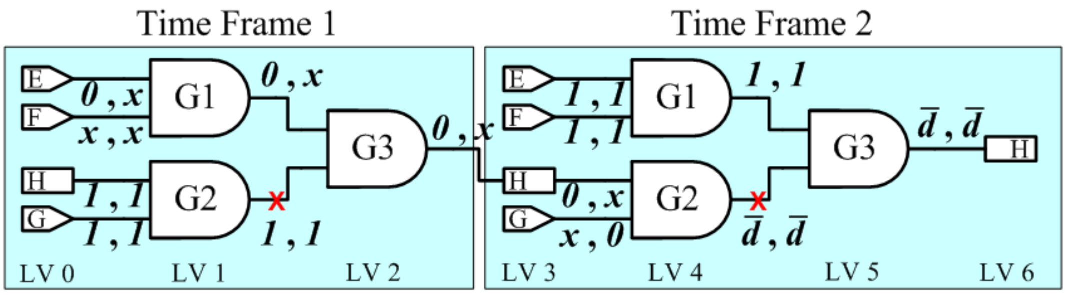

Figure 4.7: Test generation example (second backtrace and then propagation).

4.2.2 Block-Level Parallel Test Generation for a Kernel

Fig. 4.8 illustrates the block-level parallelism of t blocks generating test patterns simultaneously. Each block handles a logic level and each thread in a block handles a gate. Each x represents a target fault. The i_{th} thread in the block is shown as th_i in the figure. For example in Fig. 4.8, block 0 handles logic level 4. In block 0, th_0 and th_1 handle the upper and the lower gate in logic level 4, respectively. The number of threads needed in a block is equal to the maximum number of gates in a logic level. If the number of gates in a logic level is larger than the maximum threads allowed in a block, multiple blocks are assigned to the logic level. Each block has a copy of signals and performs test generation for different faults independently. As shown in Fig. 4.8, block 0 is performing backtrace and block 1 is performing propagation. All blocks are synchronized after all threads are finished.

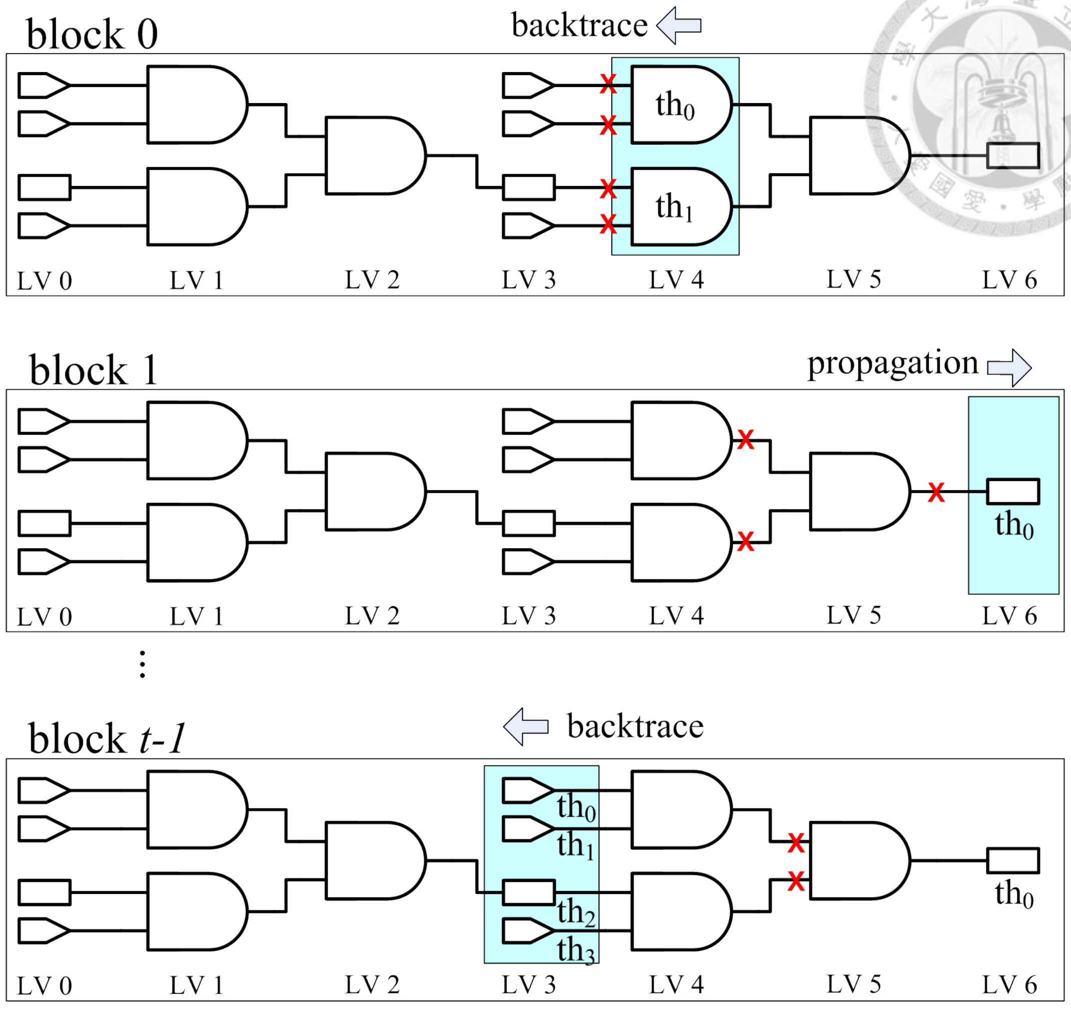

Figure 4.8: Test generation block-level parallelism.

Two variables, *currentLevel* and *currentStatus*, are stored in the shared memory of each block. The *currentLevel* represents the current logic level the block is executing. The *currentStatus* indicates the four status of the block (namely *initialize*, *backtrace*, *propagation*, and *finished*). The *currentLevel* and *currentStatus* are updated by the last thread in a block. The *currentLevel* is incremented every time when the block is propagating signals and is decremented when the block is backtracing signals. The *currentStatus* is set to backtrace when a block propagates to outputs and is set to propagation when a block backtraces to inputs. If all clones have at least one d or $\bar{d}$ at any output, *currentStatus* is set to finished. Take Fig. 4.8 for example, th_1 of block 0 updates *currentLevel* to 3, th_3 of block t-1 updates *currentLevel* to 2, and th_0 of block 1 updates *currentStatus* to *backtrace*.

The test generation kernel terminates when all blocks are finished or execution time reaches a user defined limit. To balance the load between different blocks, all blocks are working on faults in the same logic level at a time.

4.2.3 Device-Level Parallel Test Generation for a CUT

All transition faults are statically partitioned into groups, each of which is handled by a device. To balance the load between devices, faults within the same logic level are evenly distributed into different groups.

4.3 Fault Simulation Kernel

The fault simulation kernel simulates the patterns generated by the test generation kernel. There are two levels of parallel fault simulation. Word-level parallelism is implemented in a way similar to conventional PPSFP on CPU. This section only explains the block-level parallelism. The fault simulation kernel reuses memory space allocated for the test generation kernel, so no additional memory is required. Fig. 4.9 shows an example of the block-level parallel fault simulation kernel. In this example, seven blocks are created to simulate a pattern set generated by a block in the test generation kernel, so a total of $7t$ blocks are created to perform fault simulation for all patterns generated by t blocks. As in test generation, a block handles a logic level and each thread in a block handles a gate in the logic level. However, in fault simulation, the logic level assigned to a block is fixed. Each block simulates gates in its logic level and passes the results to the next block.

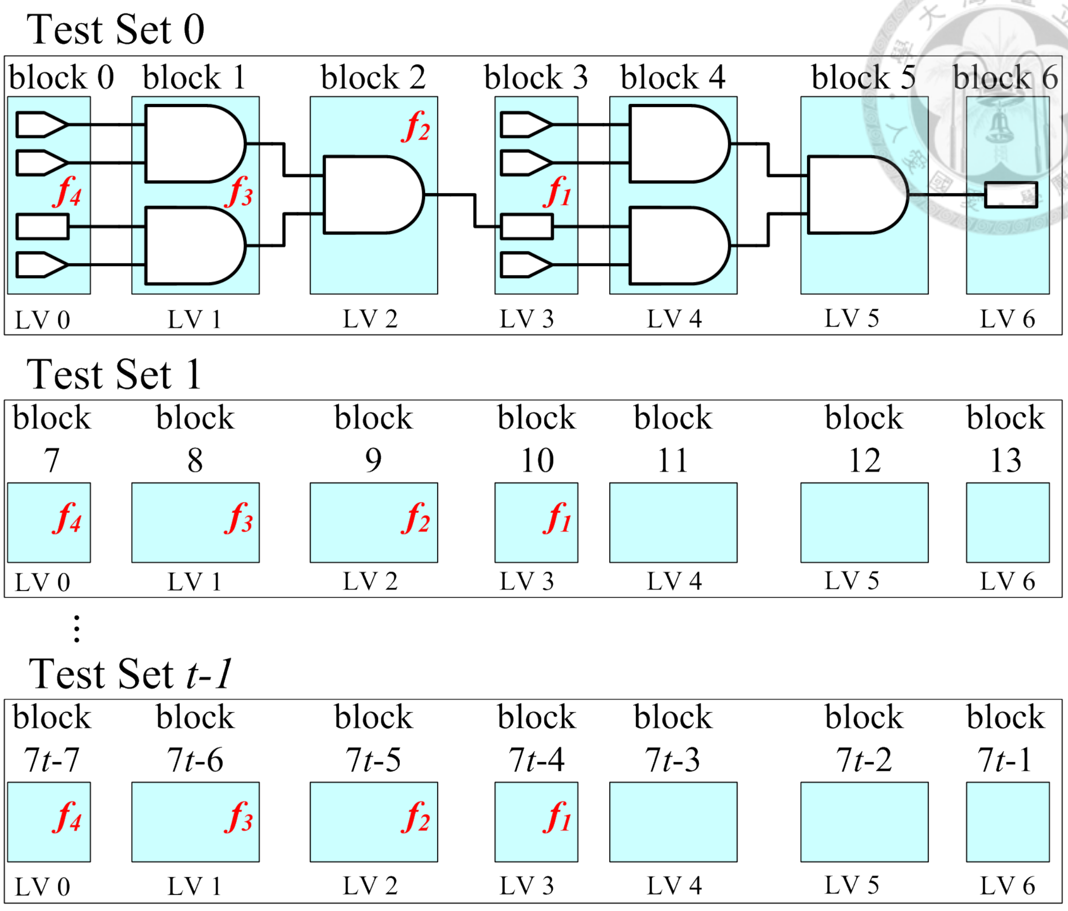

Figure 4.9: Fault simulation block-level parallelism.

The fault simulation consists of three phases. 1) Patterns in each test set are randomly filled. Parallel pattern logic simulation is then performed. 2) PPSFP is performed for all undetected faults. 3) Last logic level blocks check for d or $\bar{d}$ and update the detection number for the target fault.

Faults are shifted from low logic level to high logic level in a pipelined fashion. Take Fig. 4.9 for example, suppose there are four faults to be simulated, f_1, f_2, f_3, and f_4. The level 0 blocks (block 0, block 7, …, and block 7t-7) first simulate f_1. After each block synchronization, faults are shifted to the next block to the right. Fig. 4.9 shows the status after three block synchronizations. The fault simulation kernel ends when logic level 6 blocks finished simulating f_4.

4.4 Experimental Results

To validate the proposed GPU-based ATPG, experiments are performed on large ISCAS'89, ITC'99, and IWLS'05 benchmark circuits and a one-million gate benchmark, RSA, a cipher circuit designed by us. Results are compared with a commercial ATPG tool with multi CPU core support. The hardware specifications are summarized in Table 4.1. The GTX590 graphic card contains two devices.

Table 4.1: Platform specification

	CPU	GPU
Model	i7-2600	GTX590
#Cores	8 processors	16 Stream Multiprocessors 512 Stream Processors
Clock speed	3.4 GHz	1.22 GHz
Memory	16 GB	1.5GB

A single-core CPU-based version of our SWK algorithm is compared with the proposed GPU-based version. Table 4.2 shows the runtime (RT), test length (TL), and fault coverage (FC) of our CPU-based version and GPU-based version using one device. The last row shows the averaged results normalized with respect to those of the CPU-based version. Overall, the GPU-based version has 6.7 times of speedup compared with CPU-based version.

Table 4.3 and Table 4.4 shows the results of 8-detect transition fault test generation of a commercial tool compared with proposed GPU-based ATPG using two devices. In our test generation kernel, 64 blocks are used except leon3mp and RSA, which only uses 16 blocks due to insufficient global memory. The first two columns show benchmark names and primitive gate counts. Single-core, 4-core, and 8-core commercial ATPGs are used to generate test patterns. No significant speedup is observed beyond 8-core parallel ATPG.

Table 4.2: Compare with CPU SWK

Circuits	CPU SWK			GPU 1 device		
	RT(s)	TL	FC(%)	RT(s)	TL	FC(%)
s38417	101	1,591	98.55	21	1,602	98.58
s38584	187	3,046	92.14	22	3,138	92.25
b17	3,982	17,983	81.57	571	19,873	81.68
b18	27,903	69,750	78.62	3,687	71,691	78.65
b19	76,455	138,734	77.61	10,657	139,566	77.68
Normalize	1.00	1.00	1.00	0.15	1.04	1.00

The last row shows the averaged result normalized to that of the single-core commercial ATPG. Commercial tool results for RSA are not finished at the time of submission. We stopped the single-core commercial tool before it is finished and recorded its results in the table. The normalized numbers excludes the RSA case. Compared with CPU-based single-core commercial ATPG, our 2-device GPU-based ATPG achieved 5.9 times of speedup with 44% test length overhead. Compared to 8-core CPU-based parallel ATPG, our ATPG is 1.6 times faster.

Table 4.3: Commercial 8-detect test generation using 1 core and 4 cores

Circuits	#Gates	Commercial 1 core			Commercial 4 cores		
		RT(s)	TL	FC(%)	RT(s)	TL	FC(%)
s38417	24K	55	1,288	98.48	15	1,280	98.69
s38584	21K	86	2,715	92.36	31	2,820	92.44
b17	42K	2,614	13,572	75.61	884	16,623	81.03
b18	116K	16,237	54,041	74.08	6,397	60,711	77.76
b19	234K	41,189	108,840	73.30	14,698	123,031	77.06
leon3mp	978K	326,139	168,585	94.16	150,807	208,285	97.32
RSA	1,103K	1,429,670	145,425	95.45	705,584	171,281	97.12
Normalized	-	1.00	1.00	1.00	0.48	1.18	1.03

To evaluate the quality of our test sets, delay test coverage (DTC) is used as test quality

Table 4.4: Compare GPU SWK with commercial 8 cores 8-detect ATPG

Circuits	Commercial 8 cores			GPU SWK 2 devices		
	RT(s)	TL	FC(%)	RT(s)	TL	FC(%)
s38417	12	1,345	98.71	13	1,682	98.71
s38584	18	2,884	92.46	13	3,584	92.46
b17	693	17,424	81.76	372	20,670	81.80
b18	5,419	63,909	78.55	1,989	75,053	79.62
b19	12,106	127,461	77.55	6,797	142,471	78.73
leon3mp	106,533	229,351	98.53	68,003	268,338	98.61
RSA	526,774	183,243	97.78	387,249	197,939	98.13
Normalized	0.34	1.27	1.04	0.26	1.44	1.04

metrics. DTC calculates the effectiveness of a test set in detecting transition faults through long paths, and is used to evaluate the quality of our test patterns. DTC is defined as

$$DTC = \frac{\sum\limits_{f \in F} W_f}{|F|} \tag{4.1}$$

$$W_f = \frac{D_f^a}{D_f^s} \tag{4.2}$$

where D_f^s is the structural longest path through fault f, and D_f^a is the actual longest path through fault f sensitized by the test set. F is the set of transition delay faults and $|F|$ is the number of total faults. The upper bound of delay test coverage is equal to the transition delay fault test coverage. The larger the DTC, the better the test set is in detecting transition delay faults through long paths. Table 4.5 shows the DTC of our ATPG and 8-core commercial tool generated test sets. The last row shows the average results normalized with respect to those of commercial 8-core ATPG. The DTC of our generated test sets is higher than those of commercial ATPG but our test length is longer.

Test selection is applied to show that a compact test set selected form our generated tests remains high DTC coverage. The test selection is performed by [Chang 13]. Com-

Table 4.5: DTC comparison

Circuits	Comm. 8 cores	GPU SWK 1 device	GPU SWK 2 decives
s38417	92.45	93.89	93.92
s38584	84.48	85.70	85.75
b17	57.86	58.24	58.87
b18	60.84	62.61	63.57
b19	60.32	62.15	64.66
Normalized	1.00	1.02	1.03

mercial 8-core ATPG and our 2-device ATPG are compared in this experiment. Table 4.6 shows the DTC and test length of the patterns selected. The last row shows the averaged results normalized to that of commercial 8-core ATPG. The results show that the DTC of the test patterns selected from our ATPG is 2% higher than that of selected from the commercial 8-core ATPG with 4% test length overhead only.

Table 4.6: Test length and DTC comparison after test selection

Circuits	Comm. 8 cores		GPU SWK 2 devices	
	TL	DTC(%)	TL	DTC(%)
s38417	356	90.15	411	90.22
s38584	498	83.22	547	83.27
b17	2,793	54.40	2,878	56.70
b18	9,252	58.34	9,261	58.89
b19	18,503	57.97	19,728	60.44
Normalized	1.00	1.00	1.04	1.02

4.5 Parallelism Analysis

An experiment, which uses different number of blocks and different number threads in a block to generate 8-detect transition fault test patterns, is proposed to show the parallelism limitation of the GPU device. Table 4.7 shows the runtime of the test generation under different settings. From the table, we can observe that the runtime speedup saturated near using 16 blocks and using 128 threads per block. The reason behind this is that the GPU has reached its maxmium computation efficiency. For this GPU device, there are 16 stream multiprocessors, and within each stream multiprocessor, there are 32 stream processors (see Table 4.1). Each stream multiprocessors can switch between four blocks at the same time (hardware limitation) and each stream processor can also switch between four threads during memory access. As a result, if all threads are executed efficiently, at most 64 blocks and 128 threads per blocks can execute in parallel. Please note that this constraint is limited by the current GPU hardware. As the number of stream processors increase, the number of threads that can be executed in parallel will increase as well.

Table 4.7: Runtime analysis using different number of threads and blocks (b19)

#blocks	#threads per block			
	64	128	256	512
1	516,913	218,346	218,147	218,142
16	36,002	15,210	15,023	14,985
32	20,883	9,087	8,948	8,931
64	11,437	6,797	6,773	6,766
128	11,305	6,801	6,791	6,782

In our current architecture, the memory consumption of a block in the test generation kernel is proportional to netlist size. In Fig. 4.10, the diamond line shows the maximum number of blocks allowed with respect to different netlist size on our GPU device (1.5GB global memory). The number of blocks used in experiment is rounded down to multiples of 16 since there are 16 stream multiprocessors on our device.

The upper dashed line shows the upper bound of blocks due to the stream multiprocessors. In our experiments, using more than 64 blocks has no additional speedup so at most 64 blocks are used. The lower dashed line shows the minimum number of blocks needed to make full use of all 16 stream multiprocessors. According to Figure 10, our proposed GPU-based ATPG is scalable to approximately 1.2M gate design while maintaining 16 blocks parallel test generation, given 1.5GB global memory. When the size of design increases beyond 1.2M gates, the number of blocks decreases below 16.

The GPU hardware is improving every year in terms of memory capacity, number of cores, and clock speed. Our tool can be easily scaled to larger designs with the hardware improvement on GPU.

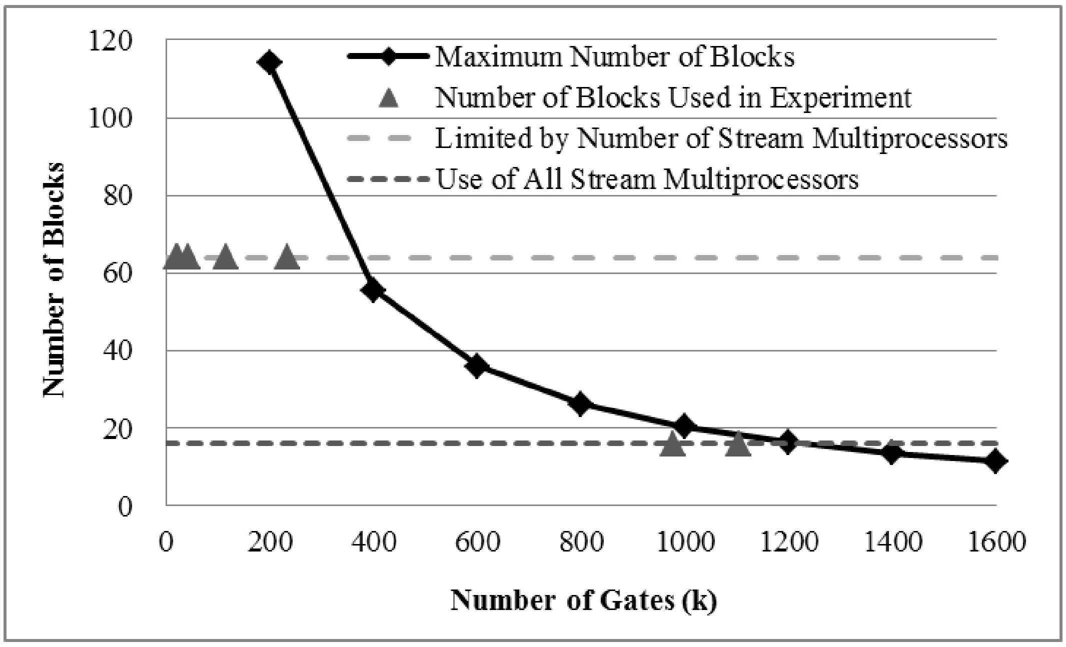

Figure 4.10: GPU scalibity analysis.

Fig. 4.11 shows the runtime after applying memory optimization techniques to benchmark circuit b19 using single device. Technique A is allocating netlist in cached texture memory. Technique B is coalesced access to global memory plus technique A. Technique C is using zero-copy to transfer patterns from global memory to CPU main memory plus technique B. By comparing techniques A, B, and C with the version without optimization, runtime is reduced by 7%, 25%, and 32%, respectively.

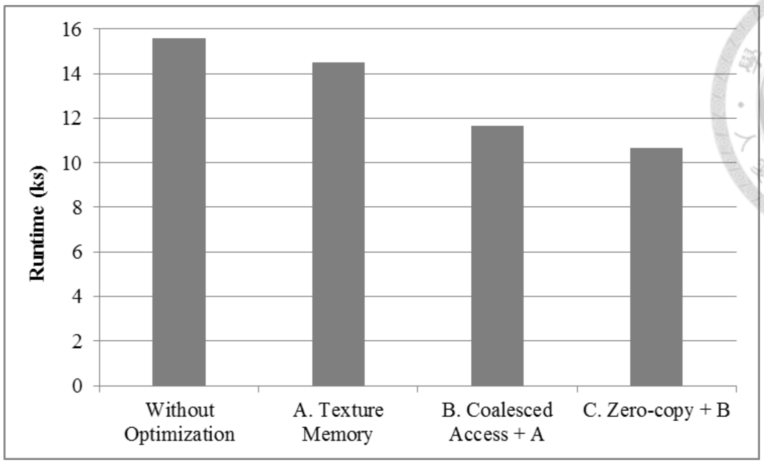

Figure 4.11: Runtime of memory access optimization techniques (b19).

4.6 Summary

A GPU-based massively parallel ATPG algorithm is proposed to accelerate SWK algorithm. Three-level parallelism is exploited in the test generation kernel. Experimental results show that our GPU-based ATPG achieves 5.6 times speedup over single-core CPU-based commercial ATPG. The DTC of our generated test patterns is slighter higher than those of 8-core CPU-based commercial ATPG. To the best of our knowledge, this is the first proposed GPU-based ATPG algorithm.

Chapter 5

High Quality Test Generation

Three extensions to the GPU-based SWK framework are proposed to deal with timing-aware and cell-aware issues. The first extension proposed a novel GPU-based test selection scheme to select test patterns from the generated N-detect test set. A memory efficient dictionary is proposed to store patterns that sensitize long paths. Then followed by a greedy heuristic to select a compact and high quality test set. The second extension proposed a new approach to perform timing-aware test generation. Unlike traditional timing-aware ATPG, which targets a specific long paths, the proposed ATPG is capable of targeting N random long paths simultaneously. The *weighted split vector* is introduced to replace the original split vector to guide ATPG through long activation and propagation paths. The generated timing-aware patterns have quality almost the same as those generated by commercial timing-aware ATPG while the ATPG runtime and the test length are much shorter. The third extension identifies a possible FinFET cell-internal open defect that would cause small extra delays. *Gate exhaustive transition* ATPG is proposed to generate all gate input transition combinations to detect these kind of defects. Results show that the achieved coverage is better than commercial 30-detect test patterns.

5.1 Test Selection for Small Delay Defects

It is shown in Chapter 4 that our generated N-detect test patterns offers higher DTC compared to that of generated by the commercial ATPG. A test selection scheme is proposed to select a compact high quality test set from N-detect test patterns for small delay defects (SDD) detection. There are two important features in this scheme. The first features is *upper and lower bound (UB/LB) analysis* that quickly decides whether a particular pattern drops a fault or not. If the static UB/LB meets certain conditions, then the fault can be dropped or undropped without detailed path length calculation. The second is that this algorithm builds only a partial fault dictionary on those faults that have not been dropped before (likely to be *hard faults*) so that the dictionary size is very small. The reason behind this is is that if a test pattern does not detect a hard fault, it is not likely to be selected in the minimum test set. Top-off patterns are added to detect those transition faults that cannot meet the fault dropping criterion

Algorithm 9 shows the overall flow of this technique. Given three inputs: a test set T, a fault list FL, and a DSM threshold δ. First, the upper and low bounds of arrival time (AT) and propagation time (PT) are calculated in the preprocess stage. During the fault simulation, a partial dictionary D is built. Building this dictionary is fast because static UB/LB are used to reduce the number of simulation and path length calculation. A greedy heuristic is used to select a small set, T^*. At the end, top-off patterns T^{**} are selected to detect those transition faults that cannot be dropped by the given DSM threshold. This can be simply done by repeating the above process with $\delta = 1$. Finally, the selected test set T^s is equal to the union of T^* and T^{**}.

Algorithm 9 Test Selection

```
1: procedure TestSelect($T$, $FL$, $\delta$)
2:     Static UB/LB anlysis
3:     $D \leftarrow$ BuildDictionary($T$, $FL$, $\delta$)
4:     $T^* \leftarrow$ GreedySelect($T$, $FL$, $\delta$)
5:     $FL^* \leftarrow$ undropped fault list
6:     $D^* \leftarrow$ BuildDictionary($T$, $FL^*$, $\delta$)
7:     $T^{**} \leftarrow$ GreedySelect($T$, $FL^*$, $\delta$)
8:     $T^s \leftarrow T^* \cup T^{**}$
9:     return $T^s$
10: end procedure
```

5.1.1 Fault Dropping Criterion

A transition fault is *detected* if it is excited and the faulty effect is propagated to any output(s) along any path(s). Traditionally, a fault is dropped as soon as it is detected by a test pattern. For SDD, a fault is dropped only if it is detected by a test pattern if its delay slack margin (DSM) is smaller than a user defined threshold δ [Lin 06].

$$DSM = \frac{PD_f^s - PD_f^a}{T_{TC} - PD_f^a} < \delta \tag{5.1}$$

where PD_f^s is the structural longest path through fault f, PD_f^a is the actual longest path through fault f sensitized by the test set. T_{TC} is the test clock period. It is easily observed that $T_{TC} \geq PD_f^s \geq PD_f^a$. δ is a user defined fault dropping threshold between zero and one. δ represents ratio of the detectable fault size (numerator) to the slack of sensitized path through the fault (denominator). If δ is set to one, which is equivalent to traditional transition fault dropping criterion, a fault is dropped as soon as it is detected. Smaller δ value means that PD_f^a is closer to PD_f^s, and thus more stringent fault dropping criterion. The default δ is set to 0.6, which is decided based on the tradeoff between test quality and test length in the experiments.

5.1.2 Upper and Lower Bound Analysis

Arrival time UB/LB ($AT_{UB/LB}$) and propagation time UB/LB ($PT_{UB/LB}$) are calculated to reduce the number of actual path delay calculations needed during the selection process. For each net z, its static upper and lower bounds of PT from z to each *structurally-reachable* output is calculated. Their definitions are shown as follows.

$$PT_{UB}(z, \omega) = \text{upper bound of } PT \text{ from net } z \text{ to output } \omega$$

$$PT_{LB}(z, \omega) = \text{lower bound of } PT \text{ from net } z \text{ to output } \omega$$

where ω can be either a primary output or a pseudo primary output that is in the fan-out cone of net z. Static PT_{UB} and PT_{LB} can be calculated level by level in a backward direction, from output to input. For a gate input i and gate output z,

$$PT_{UB}(i, \omega) = PT_{UB}(z, \omega) + d_i \tag{5.2}$$

$$PT_{LB}(i, \omega) = PT_{LB}(z, \omega) + d_i \tag{5.3}$$

where d_i is the gate delay from gate input i to output z. For a fan-out stem z, $PT_{UB}(z, \omega)$ is equal to the maximum $PT_{UB}(z, \omega)$ among all fan-out branches i and $PT_{LB}(z, \omega)$ is equal to the minimum $PT_{LB}(z, \omega)$ among all fan-out branches i.

$$PT_{UB}(z, \omega) = MAX_{b \in fobranch}(PT_{UB}(b, \omega)) \tag{5.4}$$

$$PT_{LB}(z, \omega) = MIN_{b \in fobranch}(PT_{LB}(b, \omega)) \tag{5.5}$$

With static $PT_{UB/LB}$, we can dynamically estimate upper and lower bounds of sensitized path length for a test pattern. Given a fault f at net z and a test pattern p, $PD_{UB}(f,p)$ is $AT(z,p)$ plus the maximum PT_{UB} from z to all detecting outputs. $AT(z,p)$ is the arrival time from launching inputs to net z. $PD_{LB}(f,p)$ can be calculated in a simlilar way except that it is the minimum PT_{LB}.

$$PD_{UB}(f,p) = AT(z,p) + MAX_{\omega \in detectingoutputs}(PT_{UB}(z,\omega)) \tag{5.6}$$

$$PD_{LB}(f,p) = AT(z,p) + MIN_{\omega \in detectingoutputs}(PT_{LB}(z,\omega)) \tag{5.7}$$

Please note that $PD_{UB}(f,p)$ and $PD_{LB}(f,p)$ are pessimistic since they are based on static structural information. Dynamic $DSM_{UB}(f,p)$ and $DSM_{LB}(f,p)$ can be simply derived from $PD_{LB}(f,p)$ and $PD_{UB}(f,p)$, respectively. Please note that larger PD corresponds to smaller DSM.

$$DSM_{UB} = \frac{PD_f^s - PD_{LB}(f,p)}{T_{TC} - PD_{LB}(f,p)} \tag{5.8}$$

$$DSM_{LB} = \frac{PD_f^s - PD_{UB}(f,p)}{T_{TC} - PD_{UB}(f,p)} \tag{5.9}$$

Comparing $DSM_{UB/LB}(f,p)$ with the fault dropping threshold δ, there are three possible outcomes. If $DSM_{UB}(f,p)$ is smaller than δ, fault f is definitely dropped by pattern p. If $DSM_{LB}(f,p)$ is greater than or equal to δ, fault f is definitely not dropped by pattern p. In either case, exact $DSM(f,p)$ calculation is not needed. Otherwise, there is no conclusion whether f can be dropped by p or not so an exact DSM calculation is required.

5.1.3 Build Dictionary

Algorithm 10 shows the pseudo code of BuildDictionary function. Initially end_{undrop} and end_{all} both point to the end of fault list. For each test pattern p in the test set T, perform a single logic simulation on p. Move the pointer f_{ptr} to the head of the fault list FL. Assume that a 64-bit simulator is used. Let F_{64} be the set of 64 faults on the right of f_{ptr} excited by pattern p —a slow-to-fall/rise fault is excited by a falling/rising transition. Perform a parallel fault simulation on F_{64}. For each detected fault f, if $DSM_{LB}(f, p)$ is larger than or equal to δ, then f cannot be dropped so no action is taken. If DSM is smaller than δ, then f can be dropped so we add pattern p into dictionary D.

Algorithm 10 Build Dicationary

```
1: procedure BuildDictionary($T$, $FL$, $\delta$)
2:     $D \leftarrow \emptyset$
3:     $end_{undrop} \leftarrow$ end of $FL$
4:     $end_{all} \leftarrow$ end of $FL$
5:     for each pattern $p$ in $T$ do
6:         logic simulation for $p$
7:         $f_{ptr} \leftarrow$ head of $FL$
8:         while $f_{ptr}$ is at left of $end_{undrop}$ do
9:             $F_{64} \leftarrow$ 64 exicited faults to the right of $f_{ptr}$
10:            parallel fault simulation for $F_{64}$
11:            for each detected fault $f$ in $F_{64}$ do
12:                calculate $DSM_{LB}(f,p)$ and $DSM_{UB}(f,p)$
13:                if $DSM_{LB}(f,p) \geq \delta$ then
14:                    next $f$
15:                else if $DSM_{UB}(f,p) < \delta$ then
16:                    $D \cup entry(f,p)$
17:                    next $p$
18:                end if
19:                calculate exact $DSM(f,p)$
20:                if $DSM(f,p) < \delta$ then
21:                    $D \cup entry(f,p)$
22:                    next $p$
23:                end if
24:            end for
25:        end while
26:    end for
27:    return $D$
28: end procedure
```

5.1.4 Greedy Selection

Algorithm 11 shows the algorithm to select a compact test set given the partial dictionary D. Minimum test pattern selection is a minimum set covering problem (NP-hard), so a two-stage greedy heuristic is implemented to find a near optimal test set. In the first stage, essential patterns that drop unique faults in the dictionary are selected. All the faults dropped by the selected patterns are removed from the fault list. In the second stage, a greedy algorithm that iteratively selects the pattern that drops the most number of undropped faults is selected.

Algorithm 11 Greedy Select

```
procedure GreedySelect($D$, $FL$)
    $T^* \leftarrow \emptyset$
    for each pattern $p$ in $D$ do
        if $p$ drops any unique fault then
            $T^* \cup p$
            remove faults dropped by $p$ from $FL$
            remove $p$ from $D$
        end if
    end for
    repeat
        calculate number of dropped faults for each $p$ in $D$
        $p \leftarrow$ pattern that drops most faults in $FL$
        $T^* \cup p$
        remove faults dropped by $p$ from $FL$
        remove $p$ from $D$
    until no more fault dropped
    return $T^*$
end procedure
```

5.1.5 Experimental Results

To validate the proposed test selection scheme, experiments are performed on large ISCAS'89 as well as ITC'99 benchmark circuits. The circuits are mapped to TSMC 0.13 μm technology and then placed and routed by commercial tools. Experimental results show runtime (RT), DSM, SDQL, and test length (TL) comparison between timing-unaware 1-detect commercial ATPG, timing-unaware 8-detect commercial ATPG, timing-aware commercial ATPG, and the proposed test selection scheme. All commercial ATPG used eight CPU cores.

Table 5.1 shows the benchmark circuit sizes and commercial timing-unaware 1-detect and timing-unaware 8-detect commercial ATPG. In order to achieve minimal test length, the commercial tool tries to generate launch-on-shift patterns first and then generate launch-on-capture patterns for the remaining faults that are not detected by the launch-on-shift patterns. Due to the slow shift clock, launch-on-shift patterns are not considered when calculating the DSM. SDQL was simulated using the same commercial tool with the following parameters: $a = 1.58 \times 10^{-3}$, $b = 2.1 \times 10^{-3}$, $\lambda = 4.96 \times 10^{-6}$. The default δ threshold was set to 0.6. The results are normalized with respect to timing-unaware 1-detect ATPG.

Table 5.1: Commercial timing-unaware test generation

Circuit	Comm. timing-unaware 1-detect				Comm. timing-unaware 8-detect			
	TL	DSM	SDQL	RT(s)	TL	DSM	SDQL	RT(s)
s38417	382	64.0	221	4	2,203	96.7	25	10
s38584	355	37.5	171	3	4,164	89.8	82	13
b17	2,164	50.2	6,779	161	22,147	79.1	1,785	258
b18	2,937	44.9	12,960	447	78,789	72.6	6,359	1,925
b19	4,880	46.8	28,256	1,073	154,475	70.9	15,826	5,786
Normalized	1.00	1.00	1.00	1.00	24.42	1.68	1.50	4.73

Table 5.2 shows a comparison between the timing-aware 8-core commercial ATPG and the proposed scheme. Both the timing-aware and N-detect patterns are launch-on-capture patterns for at speed testing. The last row shows the averaged numbers, normalized with respect to those of timing-unaware 1-detect ATPG. In terms of test length, timing-aware ATPG is 14X longer than timing-unaware test set, whereas our test length is only 3X longer (79% reduction). In terms of runtime, timing-aware ATPG is 43X longer than timing-unaware ATPG, whereas our runtime is only 4.6X longer (89% reduction). The DSM and SDQL quality of our selected test sets are very close to those of timing-aware

test sets.

Table 5.2: Compare with commercial timing-aware ATPG

Circuit	Comm. timing-aware				Proposed scheme			
	TL	DSM	SDQL	RT(s)	TL	DSM	SDQL	RT(s)
s38417	770	97.8	14	11	473	97.5	22	13
s38584	1,374	90.6	39	19	643	90.5	74	17
b17	14,642	81.2	1,545	1,499	3,148	80.2	1,713	356
b18	43,973	75.8	5,180	12,680	10,180	74.6	6,112	1,932
b19	92,133	74.5	10,675	58,135	19,979	72.4	13,183	5,874
Normalized	14.26	1.73	0.36	42.86	3.21	1.71	0.44	4.58

5.1.6 Summary

A fast and compact GPU-based test generation and selection technique for SDD is proposed. Two important features, UB/LB analysis and partial fault dictionary, are proposed to select a compact test set from the generated patterns. Experimental results show that the proposed GPU-base technique is a very promising alternative to timing-aware ATPG on multi-core CPU. Compared with multicore commercial timing-aware ATPG, the proposed scheme achieved 79% reduction in runtime and 89% reduction in test length while the test quality is similar.

5.2 Timing-Aware Test Generation

To further increase the quality of the generated patterns in detecting SDD, an improvement is made to GPU SWK. The *weighted split vectors* is proposed to replace the original *split vectors* to give longer paths higher chance to be chosen during test generation. Unlike traditional timing-aware ATPG, which typically targets a specific long path at a time, the proposed technique target N random long paths at a time. Also, the proposed technique does not need to perform path selection as most path delay based timing-aware ATPG does.

To select test patterns from such a large pool, a novel fault simulation kernel is proposed. In addition to the bound analysis in the previous Section, a novel fault dictionary, *longest path dictionary (LPDD)*, is proposed to record a fixed number of test patterns that sensitizes longest paths for each fault in a compact dictionary. LPDD successively reduces memory usage and data communication time between GPU and CPU while preserving the test pattern quality. Compared with an 8-core CPU-based timing-aware commercial ATPG, the proposed approach achieved 36% test length reductions on large benchmark circuits.

A simple illustrative example is first given here to show the concept of the idea. Fig. 5.1 shows the results of different approaches. The circuit contains two AND gates ($G1$ and $G2$) and one OR gate ($G3$) with three primary inputs (J, K and L), one primary output (M) and one scan cell (H). The numbers inside the gate represent the gate delays and the x represents the target slow-to-fall (STF) fault location. LV stands for logic level. The top right circuit shows the shortest detectable path which is often chosen by timing-unaware ATPG to reduce ATPG runtime. The bottom left circuit shows the longest detectable path which is chosen by commercial timing-aware ATPG. The bottom right circuit shows the result of the proposed ATPG, where four patterns targeting different paths are generated simultaneously. At most N paths can be targeted at the same time, where N is a user-defined number. In this case, out of the four patterns, two patterns targeting longest paths and two patterns targeting shorter paths are generated simultaneously.

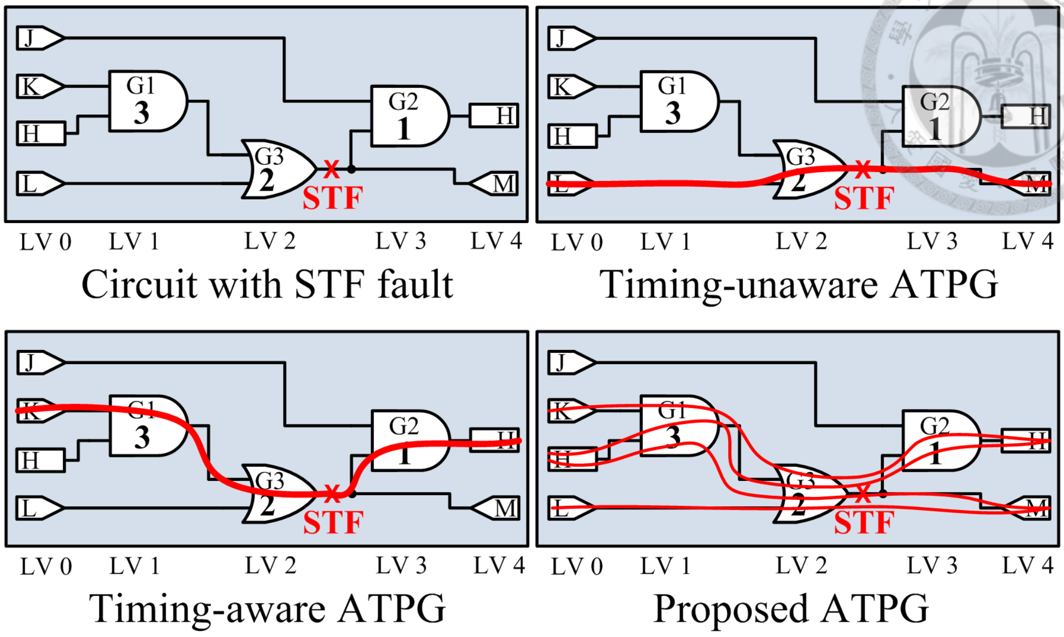

Figure 5.1: ATPG comparison.

5.2.1 Test Generation Kernel

In order to generate timing-aware patterns, *weighted split vector* is introduced to guide ATPG towards longer paths. A weighted split vector S is a vector of W bits with each bit being zero or one. Four example weighted split vectors are given below with W equals to four.

$$S_1 = 0011 \qquad \overline{S_1} = 1100$$

$$S_2 = 0101 \qquad \overline{S_2} = 1010$$

$$S_3 = 1110 \qquad \overline{S_3} = 0001$$

$$S_4 = 0010 \qquad \overline{S_4} = 1101$$

Unlike original split vectors, weight split vector can have uneven number of zeros and ones. For example, S_3 has three ones and one zero, and S_4 has three zeros and one one. The uneven number of ones and zeros represents the favor of a choice with respect to the other choice. Having more ones than zeros in a vector (such as S_3) means having higher chance of backtracing objectives through the upper fanin gate. Having more zeros than ones in a vector (such as S_4) means having higher chance of backtracing objectives through the lower fanin gate. Having same number of zeros and ones (such as S_1 and S_2) means two paths have equal chance. The number of ones and zeros in a vector is determined by the AT_{UB} of the fanin gates.

Fig. 5.2 shows an example of the different weight split vectors applied to different gates. The numbers enclosed by curly braces in the figure are the possible weighted split vectors used for each gate. The two numbers with a slash beside each gate is the AT_{UB} and PT_{UB}. Take $G1$ as an example, both of its inputs, K and H, have the same AT_{UB} (0), so a weighted split vector of same number of ones and zeros is used. For $G3$, since $G1$ has AT_{UB} of 3 and L has AT_{UB} of 0, weighted split vectors with more ones are used.

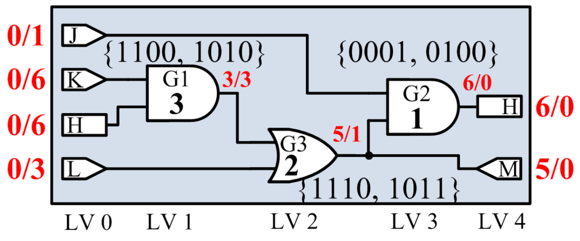

Figure 5.2: Weighted split vector example.

Fig. 5.3 to Fig. 5.7 duplicate the circuit in Fig. 5.1 into two time frames to demonstrate the test generation flow. Suppose $W = 4$ and $N = 4$, that means there are four clones targeting the $G3$ STF fault. The values of the four clones are represented italic and bold letters in the figure, e.g., $b_0b_0b_0b_0$ in the second time frame of Fig. 5.3. Since the target fault is STF, the initial objectives in time frame 1 and time frame 2 are one and zero, respectively. So all clones are set to b_1 in time frame 1 and b_0 in time frame 2.

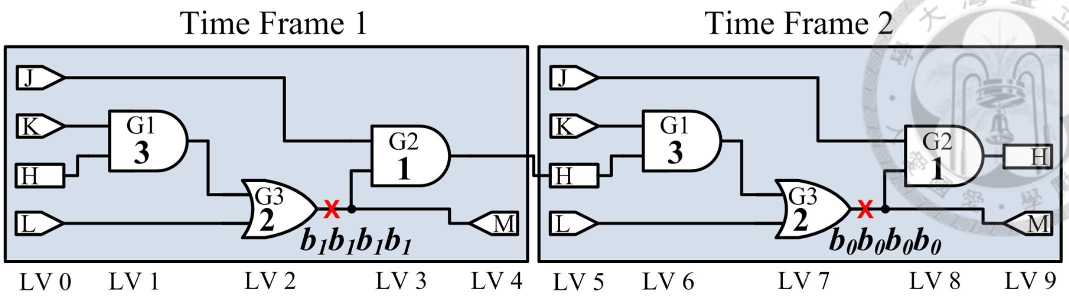

Figure 5.3: Test generation example (initial objectives).

Fig. 5.4 shows the first backtrace after Fig. 5.3. An implication backtrace is performed on $G3$ in the time frame 2 since both fanins need to be b_0 to justify the output objective b_0. An objective split is performed on G_1 in time frame 2 and $G3$ in time frame 1 (gates with arrows). For $G1$ in time frame 2, objectives are split evenly due to same AT_{UB} values (0) of K and H. For $G3$ in time frame one, however, since $G1$ has larger AT_{UB} value (3) than L (0), three objectives are backtraced towards $G1$ and only one objective is backtraced towards L.

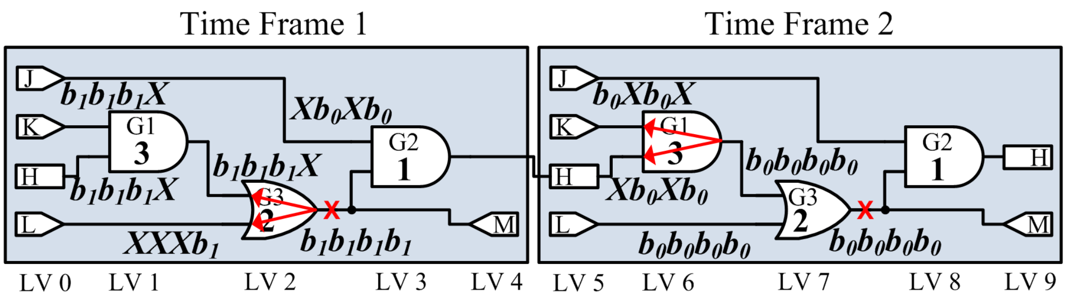

Figure 5.4: Test generation example (first backtrace).

Fig. 5.5 shows input assignment followed by propagation from input towards output. The fault effect ($\bar{d}$) is generated at the output of $G3$ in time frame 2 by all four clones. A p is generated at the output of $G3$ when one of the gate inputs is d or $\bar{d}$ while the other input is unknown, which indicates the gate is on the propagation path. For each fanout stem with value p, propagation split is performed to split p to different branches based the PT_{UB} of the fanout gates. This is similar to the objective split in the backtrace procedure.

For a fanout stem B with n fanout branches $B_1, B_2, \ldots, B_n$, each branch bitwise ANDs a weighted split vector. The following equation shows the propagation split for a stem with 2 fanout branches:

$$B_1^p = B^p \cdot S \; B_2^p = B^p \cdot \overline{S}$$

where S is a weighted split vector. The number of ones and zeros in a vector is determined by the PT_{UB} of the fanout gates. For stems with more than 2 fanout branches, the calculation is broken down to 2 branches at a time.

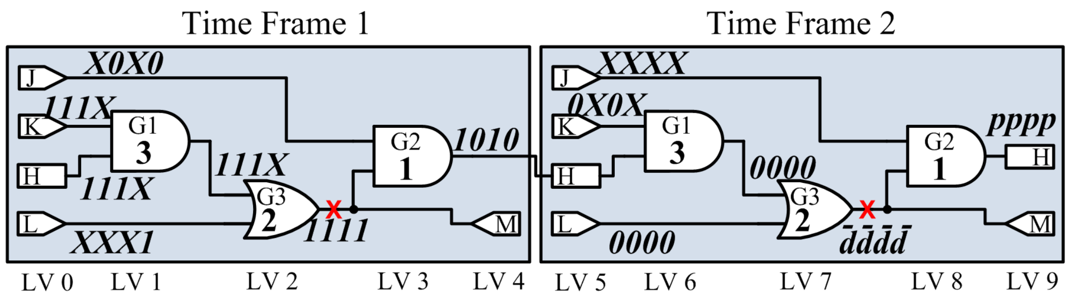

Figure 5.5: Test generation example (first propagation).

The propagation split does not happen in Fig. 5.5 since $G2$ has no fanout branches. We use the Fig. 5.6 to explain the propagation split. Suppose the value on the fanout stem $G4$ is $pppp$. The two numbers separated by a slash represent AT_{UB}/PT_{UB} of the corresponding gate output, and the vectors in curly braces are possible weighted split vectors. For $G4$, The longest possible propagation path delay through $G5$ is 4 (PT_{UB} $G5$ plus the delay of $G5$) and the longest possible one through $G6$ is 1. So $G5$ is given higher chance of propagating the fault effect by generating more ones in the weighted split vectors of $G4$. Please note that the weighted split vectors used for backtracing and the ones used for propagating are different since the AT_{UB} of the fanin gates are different from the PT_{UB} of the fanout gates.

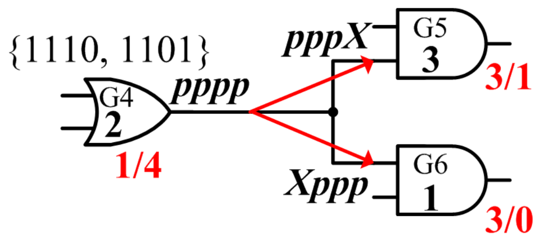

Figure 5.6: Propagation split example.

Fig. 5.7 shows the second backtrace followed by a second propagation. Input J in the time frame 2 is set to one to propagate the faulty effects to the output H. Four test patterns JKL = {(*X1X*, *100*), (*01X*, *1X0*), (*X1X*, *100*), (*0X1*, *1X0*)} and H={*111X*} are generated successfully and two of the patterns sensitize longest detectable path.

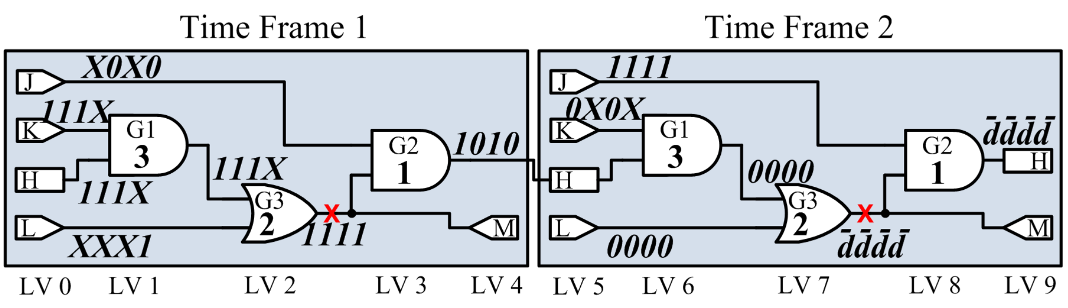

Figure 5.7: Test generation example (first propagation).

5.2.2 Fault Simulation

The timed fault simulation calculates the dynamic arrival time of gates in the fanout cone of the target fault. The simulation is event-driven so only gates on the propagation paths are calculated. After all gates in the event list have been simulated, the arrival times of the faulty outputs are compared with the value in the dictionary. If the current sensitized path delay is larger, then update the new path delay into the dictionary and record the current pattern. Fig. 5.8 shows the dynamic arrival times calculated for the gates in the fanout cone of the $G3$ STF fault. After simulation, outputs H and M have sensitized path delays of 6 and 5, respectively. Thus, 6 will be recorded in the dictionary.

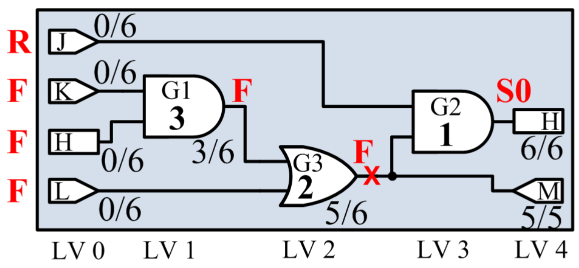

Figure 5.8: Timed fault simulation example.

Due to limited GPU memory, a complete fault dictionary is impossible to be stored on the GPU. A compact *longest path delay dictionary* (LPDD) is proposed to improve the dictionary used to test selection. For each fault, LPDD only records the *longest sensitized path delay* (LSPD) and the first U pattern IDs that sensitized it through LSPD, where U is a user-specified parameter. Table 5.3 shows an LPDD example with 6 faults (f_1, f_2, ..., f_6) simulated by m patterns (p_1, p_2, ..., p_m), where U is set to 2.

Table 5.3: LPDD example

	f_1	f_2	f_3	f_4	f_5	f_6
Delay	5	6	4	3	-	5
Pat ID	p_1	p_3	p_3	p_2	-	p_5
Pat ID	p_2	-	p_4	p_5	-	-

The greedy test selection method selects patterns from LPDD. First, it removes faults which are not detected by any pattern. Second, it selects patterns which detect faults that are not detected by any other pattern. Third, it selects patterns detecting the most number of faults and then update the LPDD. The third step is repeated until all faults are covered. Take Table 5.3 as an example, f_5 is first removed because no patterns detect the fault. Second, p_3 and p_5 are selected because they detect f_2 and f_6, and then f_3 and f_4, are removed because both faults are detected by p_3 and p_5. Third, p_1 is selected to detect only one uncovered fault, f_1. Finally, the compact test set is $\{p_1, p_3, p_5\}$.

5.2.3 Experimental Results

Table 5.4 shows the benchmark circuits and test generation results of commercial CPU-based 8-core timing-aware ATPG and the proposed timing-aware ATPG. Test length (TL), runtime (RT), and SDQL are shown for comparison. For the proposed ATPG, test length before and after test selection are shown separately. Parameter N is set to 8 and U is set to 6 in the experiment. SDQL was simulated using the same commercial tool with the following parameters: $a = 1.58 \times 10^{-3}$, $b = 2.1 \times 10^{-3}$, $\lambda = 4.96 \times 10^{-6}$. All results are summed then normalized with respect to those of the commercial timing-aware ATPG.

From Table 5.4 it can be observed that for small circuits, s38417 and s38584, our proposed ATPG and commercial 8-detect timing-unaware ATPG actually have smaller SDQL than the commercial timing-aware ATPG. This is because for most fault, eight different sensitization paths are sufficient for sensitizing at least one long path. For large circuits such as b17, b18, and b19, our test lengths are shorter than those of commercial timing-aware ATPG. Overall, compared with commercial timing-aware ATPG, the proposed ATPG achieved 36% of test length reduction and 29% of runtime reduction with almost the same SDQL.

Table 5.4: Compare with commercial timing-aware ATPG

Circuit	Comm. timing-aware			Proposed timing-aware ATPG			
	TL	RT(s)	SDQL	TL Gen	TL Sel	RT(s)	SDQL
s38417	797	12	12	1,778	1,512	49	12
s38584	1,225	16	28	3,461	1,602	92	27
b17	14,968	2,870	1,286	18,331	8,926	1,093	1,282
b18	42,336	21,537	5,058	51,847	26,518	8,630	5,059
b19	86,734	84,804	9,439	147,209	55,027	67,666	9,489
Normalized	1.00	1.00	1.00	1.52	0.64	0.71	1.00

5.2.4 Summary

A timing-aware GPU-based test generation technique for SDD is proposed. Weighted split vectors are implemented to backtrace and propagate along N random long paths for a fault so that many high quality test patterns are generated simultaneously. Generated test patterns are then fault simulated to record the first U patterns that sensitize longest path delays for a fault. A compact long path delay dictionary is proposed to record patterns that sensitize faults through long paths. Experimental data show an average of 36% reduction in test length and 29% reduction in runtime while the SDQL remains almost the same.

5.3 Gate Exhaustive Transition Test Generation

FinFET is the most popular 3D transistor in nanometer modern technologies. FinFET transistors have the benefits of low leakage over traditional planar transistors. However, research show that some defects in FinFET circuits behave differently from those in planar transistor circuits.

Our research showed that some defects in FinFET cause small extra gate delay that depends on how we lunch the transition. Fig. 5.9 shows the layout of a FinFET 2-input NAND gate with inputs AB. Because the width of FinFET cannot be adjusted, FinFET can only be sized by increasing the number of FinFETs connected in parallel. In this figure, we have up to eight fins connected in parallel.

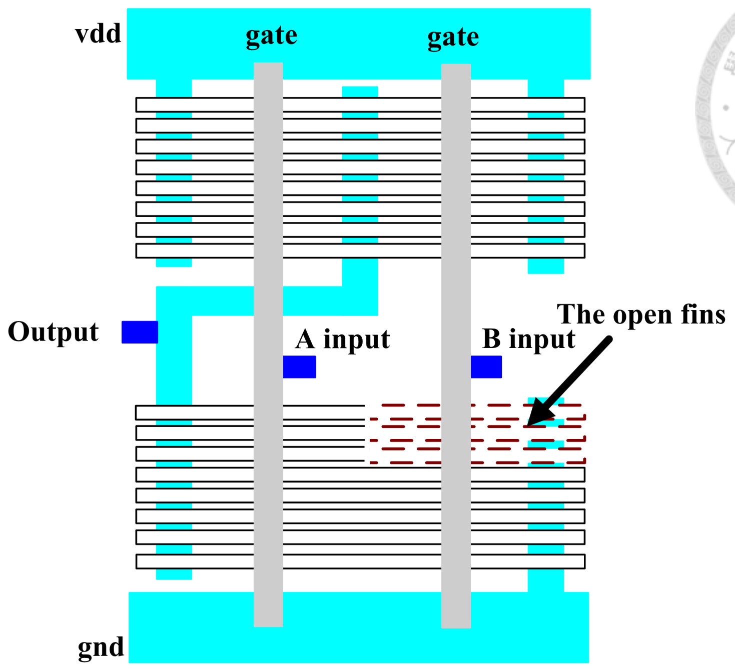

Figure 5.9: FinFET NAND layout with open defect.

Consider an open defect in n-type FinFET, which results in a weaker pull-down network. One open defect can result in multiple FinFet stuck-open. The pull-down conductance depends on the size of the open defect. We performed spice simulation on this NAND gate using FinFET model from the Predictive Technology Model (PTM) [Liu 12]. Fig. 5.10 shows that the falling delay of a defective NAND gate increases with the number of open n-type fins. On the X-axis, 0 means no fin is open (this is defect-free) and 8 means that all fins are open. This open defect causes extra falling delay at the gate output.

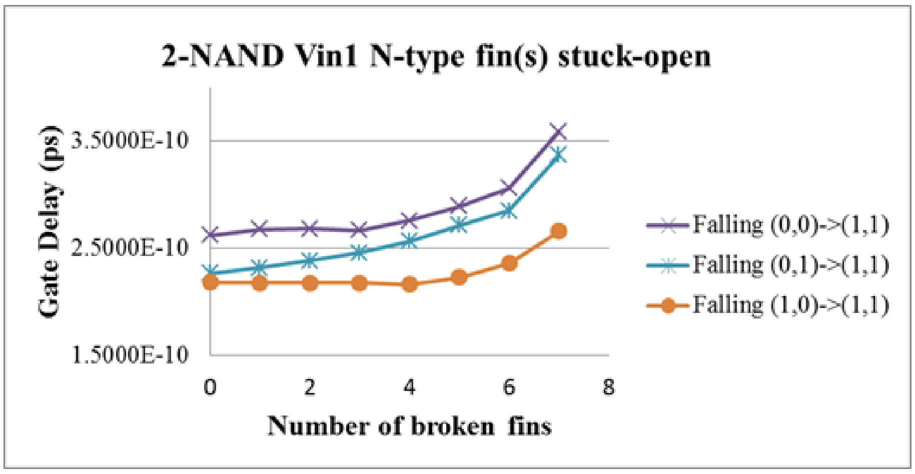

Figure 5.10: Three patterns to launch NAND output failing.

Fig. 5.10 shows three test pattern pairs that launch a falling transition (AB=00 $\rightarrow$ 11, 01 $\rightarrow$ 11, 10 $\rightarrow$ 11), where 00 $\rightarrow$ 11 has the longest delay. That means, the test pattern pair 00 $\rightarrow$ 11 is the most effective test pattern. However, traditional transition fault ATPG may not generate the test pattern pair 00 $\rightarrow$ 11. The reason can be seen in Table 5.5, where six pairs of test patterns and their detected transition faults are shown. The first test pattern (01 $\rightarrow$ 11) and second test pattern (10 $\rightarrow$ 11) detects input A slow-to-rise (STR) and input B STR faults, respectively. They also detect output C slow-to-fall (STF) fault. The third test pattern (00 $\rightarrow$ 11) does not detect any unique transition fault. Therefore, traditional transition fault ATPG may not activate the fault by P_3 (00 $\rightarrow$ 11). Even if P_3 is generated, SDD may not be detected since timing-unaware ATPG may not propagate fault effect via long paths. Although timing-aware ATPG can propagate the fault effect via long paths, it may not activate the fault by P_3, either.

Table 5.5: Patterns detecting transition faults and fault combinations

Pattern	AB V_1	AB V_2	transition fault	fault combination
P_1	01	11	A STR, C STF	C STF$_1$
P_2	10	11	B STR, C STF	C STF$_2$
P_3	00	11	A STR, B STR, C STF	C STF$_3$
P_4	11	01	A STF, C STR	C STR$_1$
P_5	11	10	A STF, C STR	C STR$_2$
P_6	11	00	C STR	C STR$_3$

Gate Exhaustive Transition ATPG is proposed to detect FinFET cell-internal defects. Two metrics, *GET Coverage* and *GET SDQL*, are also proposed to meature the quality of the generated test aptterns. As is shown in Table 5.5, a two-input NAND gate has six transition faults: A STR, A STF, B STR, B STF, C STR, and C STF. For GET ATPG, a two-input gate has only two faults C STF and C STR, each of which has three gate input combinations: C STF$_1$, C STF$_2$, C STF$_3$, C STR$_1$, C STR$_2$, C STR$_3$. These are called as six *fault combinations*. Traditional transition fault ATPG would select only three test patterns (P_3, P_4, and P_5) to detect all six transition faults. However, GET ATPG would need all six patterns P_1 P_6 to achieve 100% coverage. The GET objectives are added to the GPU-based SWK ATPG to generate all possible transtion combination for all gates and propagate the faulty effects through long paths. Unlike cell-aware ATPG, GET does not require exhaustive SPICE simulation to characterize each library cell.

5.3.1 GET Coverage and GET SDQL

For a gate i, the number of fault combinations to launch a gate out falling transition is $n_{f,i}$. The number of fault combinations to launch a gate output rising transition is $n_{r,i}$. Take a two-input NAND gate for example. There are three fault combinations to launch a rising transition at the gate output (11 $\rightarrow$ 00, 11 $\rightarrow$ 01, 11 $\rightarrow$ 10) so n_r = 3. There are also three fault combinations to launch a falling transition at gate output (00 $\rightarrow$ 11, 01 $\rightarrow$ 11, 10 $\rightarrow$ 11) so n_f = 3. Suppose there are totally G gates in the circuit. *GET coverage* is defined as follows.

$$GET\,Coverage = \frac{\sum_{i=1}^{G} n_{d,i}}{\sum_{i=1}^{G}(n_{r,i} + n_{f,i}) - n_{ut}} \times 100\% \tag{5.10}$$

where $n_{d,i}$ indicates the number of detected gate output transition faults by different fault combinations, and n_{ut} is the number of untestable fault combinations. A fault combination is untestable if its corresponding gate output transition is untestable or its gate input transition fault is untestable. For example, if the NAND gate output slow-to-fall fault is untestable, then all its three corresponding fault combinations are untestable.

Table 5.6 shows the number of traditional faults versus the number of fault combinations for an n-input NAND gate. For a three-input NAND gate, traditional transition fault has 8 faults while GET ATPG has 14 fault combinations. Generally, for an n-input NAND gate, the number of transition faults is 2×(n-1) while the number of GET fault combination is 2×(2n-1). Typically, the number of gate inputs is not large so the number of fault combinations is reasonably small.

Table 5.6: Number of faults for n-input NAND gate

n	#transition faults	#fault combinations
2	6	6
3	8	14
4	10	30
n	2×(n+1)	2×(2n-1)

Complex gates are handled in a similar way. Fig. 5.11 shows the output STF fault combinations for an AOI (AND OR INV) cell. R represents a rising transition and F represents a falling transition. *1* represents a static one and *0* represents a static zero. Generally, if we break down a complex gate into a series of n 2-input elementary gates, the number of fault combinations grows exponentially: 3^n. Fortunately, the number n is generally small (less than 3 in our library) so the number of fault combinations is acceptable.

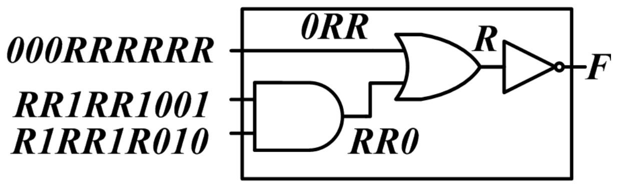

Figure 5.11: Nine fault combinations for AOI complex cell.

The *GET SDQL* can be calculated in a similar way as traditional SDQL.

$$GET\,SDQL = \sum_{i=1}^{K} \int_{T_{mgn}}^{T_{det}} F(s)ds \tag{5.11}$$

where K is the total number of fault combinations in the circuit. Traditional transition fault ATPG calculates SDQL for each transition fault, whereas GET ATPG calculates SDQL for each fault combination. An in-house simulation is implemented to calculate the GET SDQL. Since it is hard to prove longest true paths for a fault combination, structural longest path is used to replace longest true path when calculating T_{mgn}. In addition, the number of GET fault combinations is larger than the number of transition faults. Hence, the value of GET SDQL is larger than traditional SDQL.

5.3.2 Test Generation

Two features are added to the original GPU-based SWK ATPG: 1) multiple initial objectives for each fault combination; and 2) activate and propagate faulty effect through long paths. The second feature is implemented using the weighted split vector mentioned in the previous section. Fig. 5.12 explains how objectives are initialized. In this example, the target fault is STF at the output of a NAND gate. Suppose four patterns are generated simultaneously. Each objective represents a test pattern. The gate in the upper figure shows objective initialization of the original SWK ATPG, which are four falling objectives at the output of the gate. After the first split, only two of three fault combinations, (R,*1*) and (*1*,R), are initialized so SWK ATPG cannot assign all three possible combinations. The three gates in the lower firgure shows multiple objective initialization for GET ATPG. Since GET ATPG treats all fault combinations independently, all objectives for the three fault combinations will be initialized.

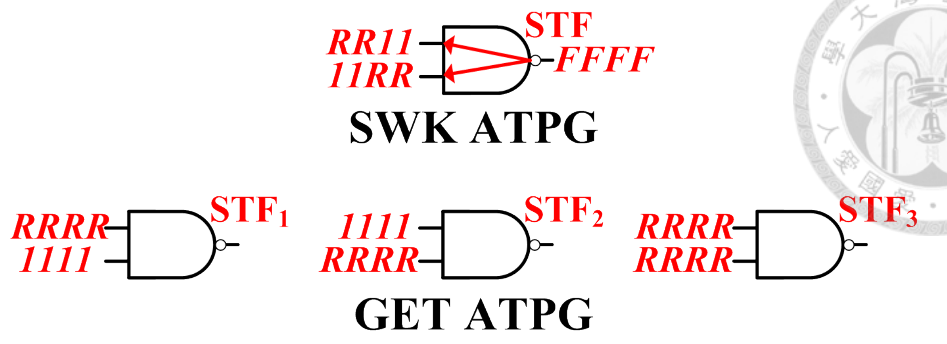

Figure 5.12: Multiple fault combination initial objectives.

5.3.3 Experimental Results

To validate the proposed GET ATPG, the results are compared with CPU-based multi-core commercial ATPG. Large benchmark circuits from ISCAS'89 and ITC'99 are mapped to TSMC 0.13μm technology. Table 5.7 shows the the number of transition faults (TF), the number of fault combinations (FC), and the number of untestable fault combinations (UT). The number of untestable fault combination is proven by commercial ATPG. As can be seen in the table, the number of FC is two to three times the number of TF. Also, the number of UT accounts for 7% of the number of FC.

Table 5.7: Circuit information

Circuit	#TF	#FC	#FC/#TF	#UT	#UT/#FC
s38417	47,686	92,786	1.94	1,862	0.02
s38584	41,434	109,122	2.63	10,750	0.09
b17	65,086	216,258	3.32	22,109	0.10
b18	231,412	727,600	3.14	66,944	0.09
b19	466,916	1,462,968	3.13	101,647	0.06
Average	170,507	521,747	2.83	40,662	0.07

CPU-based multicore commercial ATPG are used to compare with the proposed GET ATPG. Test patterns generated by commercial 6-detect transition fault ATPG and commercial timing-aware ATPG are simulated by an in-house simulator to calculate the GET coverage and GET SDQL. In Table 5.8, test length (TL), runtime (RT), and the proposed two test metrics are shown. The results of different circuits are summed and normalized with respect to those of timing-unaware 6-detect transition fault test sets, generated by a commercial ATPG on an 8-core CPU. Table 5.9 shows the result of the proposed GET ATPG. As can be observed from the table, with approximately the same test length, GET ATPG achieved 4% and 2% better GET coverage and GET SDQL than 6-detect ATPG, respectively. Also, compared with timing-aware test sets, GET test sets improved GET coverage and GET SDQL by 4% and 1%, respectively.

Table 5.8: Commercial ATPG results

Circuit	Comm. timing-unaware				Comm. timing-aware			
	TL	RT(s)	GET Cov	GET SDQL	TL	RT(s)	GET Cov	GET SDQL
s38417	1,212	7	81.10	5,723	797	12	79.80	5,767
s38584	2,144	10	75.36	12,676	1,225	16	73.43	12,749
b17	12,131	206	55.46	92,708	14,968	2,870	57.03	91,692
b18	42,015	1,496	56.26	202,525	42,336	21,537	57.27	198,958
b19	82,972	4,558	54.36	424,781	86,783	84,804	55.76	418,055
Norm.	1.00	1.00	1.00	1.00	0.56	13.67	1.00	0.99

Table 5.10 shows GET coverage of increasing number of N-detect transition fault test sets for circuit b17. It can be seen that, our ATPG is approximately the same in length as 5-detect ATPG but the quality is better. As N increases, GET coverage increases very slowly because N-detect ATPG does not specifically focus on fault activation and fault effect propagation. It requires very large N for traditional N-detect test sets to achieve the same GET coverage and GET SDQL as the proposed GET ATPG.

Table 5.9: GET ATPG results

Circuit	GET ATPG			
	TL	RT(s)	GET Cov	GET SDQL
s38417	2,626	63	82.96	5,546
s38584	4,147	117	78.32	12,533
b17	14,240	1,294	58.37	91,333
b18	46,021	13,299	58.70	197,211
b19	94,378	924,89	57.17	414,282
Normalized	0.61	20.24	1.04	0.98

Table 5.10: Commercial N-detect quality analysis

ATPG	GET Cov	GET SDQL	TL
1-detect	49.57	94,517	2,951
2-detect	53.46	93,384	6,295
4-detect	55.46	92,751	12,004
6-detect	56.40	92,415	17,368
8-detect	56.80	92,211	21,973
10-detect	57.03	92,064	26,774
15-detect	57.59	91,790	37,089
20-detect	57.92	91,612	47,248
30-detect	58.25	91,448	65,016
GET	58.37	91,333	14,240

5.3.4 Summary

GET ATPG is proposed to target FinFET cell-internal defects. Two test metrics, GET coverage and GET SDQL, are proposed to measure the test quality of the generated test patterns. All objetives of the fault combinations can be satisfied through the proposed multiple initialized objectives concept. GET ATPG combines the merits of cell-aware and timing-aware ATPG. It first initializes the objectives of the fault combinations and then propagates the faulty effect through long paths. Given approximately the same test length, the proposed ATPG has 4% higher GET coverage than N-detect transition fault

ATPG. It takes very large N (> 30) to achieve the same GET coverage as our ATPG. Also, given almost the same GET SDQL, our proposed ATPG has higher GET coverage than timing-aware ATPG.

5.4 Summary

Three experiments in this chapter demonstrated the GPU-based SWK is capable of generating high quality test patterns quickly. The first experiment proves the generated N-detect test patterns have good SDD detection ability by showing high DTC and low SDQL. Through upper and lower bound analysis and test selection, the proposed scheme achieved 79% reduction in runtime and 89% reduction in test length while the test quality is similar to that of timing-aware ATPG.

The second experiment added the weighted split vector feature to the original GPU-based SWK ATPG so that N random long paths are targeted simultaneously. A compact long path dictionary is an memory efficient dictionary that records only a number of patterns that sensitize faults through long paths. Experimental data show an average of 36% reduction in test length and 29% reduction in runtime while the SDQL remains almost the same.

The third experiment proposed gate exhaustive transition ATPG to detect each fault combinations in the circuit. Multiple fault combination objectives are initialized simultaneouly at the fault site. GET ATPG combines the merits of cell-aware and timing-aware ATPG and does not need SPICE library simulation. Given approximately the same test length, the proposed ATPG has 4% higher GET coverage than N-detect transition fault ATPG. It takes very large N (> 30) to achieve the same GET coverage as our ATPG. Also, given almost the same GET SDQL, our proposed ATPG has higher GET coverage than timing-aware ATPG.

Chapter 6

Conclusion

To deal with the two critical issues, complex defect behavior and long ATPG runtime, in testing modern chips, a GPU-based ATPG framework is proposed. Unlike CPU-based ATPG which relies on fast serial decision makeing and generates one test pattern at a time, the proposed framework is capable of targeting multiple test objectives and multiples faults at the same time. This framework a completely new approach to the current test issues. To the best of our knowledge, this is the first proposed GPU-based ATPG algorithm.

The framework implements three levels of parallelisms: device-level fault partitioning, block-level circuit partitioning, and word-level search space partitioning. The result is a massively paralleled algorithm which can generate thousands of patterns simultaneously. Such parallelism has not been achieved on traditional CPU-based ATPG.

The core of the framework is the SWK parallel ATPG algorithm, which can generate test patterns that meet multiple objectives. SWK uses random split to convert decisions into parallel bitwise logic operations so that multiple objectives can be tried at the same time. Four test metrics, N-detect, PAN, BCE, and GE are targeted simultaneously by SWK in the experiments. Results show that PAN coverage of SWK test patterns is 10% to 20% higher than that of high quality 50-detect ATPG patterns while the former is shorter. A GPU-based massively parallel algorithm is then proposed to accelerate SWK algorithm. Experimental results show that our GPU-based ATPG achieves 5.6 times speedup over single-core CPU-based commercial ATPG. The DTC of our generated test patterns is slighter higher than those of 8-core CPU-based commercial ATPG.

Three extensions were proposed on the GPU-based SWK framework to deal with timing-aware issues and cell-aware issues. Results shown that the proposed framework achieved higher quality, shorter test length, and shorter runtime compared with state-of-the-art CPU-based commercial ATPG. Overall, this dissertation provided a whole new massively parallel approach to the test issues at hand and proved through experiments that the generated test patterns are compact and high quality.

Reference

[Aitken 99] R. C. Aitken. "Extending the pseudo-stuck-at fault model to provide complete iddq coverage". *Proceedings IEEE VLSI Test Symposium*, 1999:128–134.

[Alampally 11] S. Alampally, R. Venkatesh, P. Shanmugasundaram, R. A. Parekhji, and V. Agrawal. "An efficient test data reduction technique through dynamic pattern mixing across multiple fault models". *Proceedings IEEE VLSI Test Symposium*, 2011:285–290.

[Cai 10] X. Cai, P. Wohl, J. A. Waicukauski, and P. Notiyath. "Highly efficient parallel atpg based on shared memory". *Proceedings IEEE International Test Conference*, 2010:1–7.

[Chang 13] C.-Y. Chang, K.-Y. Liao, S.-C. Hsu, J.-M. Li, and J.-C. Rau. "Compact test pattern selection for small delay defect". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(6), 2013:971–975.

[Cho 05] K. Y. Cho, S. Mitra, and E. McCluskey. "Gate exhaustive testing". *Proceedings IEEE International Test Conference*, 2005:770–777.

[Friedman 74] A. D. Friedman. "Diagnosis of short-circuit faults in combinational circuits". *IEEE Transactions on Computers*, 1974:746–752.

[Fujiwara 83] H. Fujiwara and T. Shimono. "On the acceleration of test generation algorithms". *IEEE Transactions on Computers*, 100(12), 1983:1137–1144.

[Galiay 80] J. Galiay, Y. Crouzet, and M. Vergniault. "Physical versus logical fault models mos lsi circuits: Impact on their testability". *IEEE Transactions on Computers*, 29(6), 1980:527–531.

[Goel 81] P. Goel. "An implicit enumeration algorithm to generate tests for combinational logic circuits". *IEEE Transactions on Computers*, 100(3), 1981:215–222.

[Goel 09] S. K. Goel, N. Devta-Prasanna, and R. P. Turakhia. "Effective and efficient test pattern generation for small delay defect". *Proceedings IEEE VLSI Test Symposium*, 2009:111–116.

[Gulati 08] K. Gulati and S. P. Khatri. "Towards acceleration of fault simulation using graphics processing units". *Proceedings IEEE Design Automation Conference*, 2008:822–827.

[Gupta 04] P. Gupta and M. S. Hsiao. “Alaptf: A new transition fault model and the atpg algorithm”. *Proceedings IEEE Internation Test Conference*, 2004:1053–1060.

[Hamzaoglu 98] I. Hamzaoglu and J. Patel. “New techniques for deterministic test pattern generation”. *Proceedings IEEE VLSI Test Symposium*, 1998:446–452.

[Hapke 09] F. Hapke, R. Krenz-Baath, A. Glowatz, J. Schloeffel, H. Hashempour, S. Eichenberger, C. Hora, and D. Adolfsson. “Defect-oriented cell-aware atpg and fault simulation for industrial cell libraries and designs”. *Proceedings IEEE International Test Conference*, 2009:1–10.

[Jahangiri 05] J. Jahangiri and D. Abercrombie. “Value-added defect testing techniques”. *IEEE Transactions on Design and Test of Computers*, 22(3), 2005:224–231.

[Kirkland 87] T. Kirkland and M. R. Mercer. “A topological search algorithm for atpg”. *Proceedings IEEE Design Automation Conference*, 1987:502–508.

[Klenke 92] R. H. Klenke, R. D. Williams, and J. H. Aylor. “Parallel-processing techniques for automatic test pattern generation”. *IEEE Transactions on Computers*, 25(1), 1992:71–84.

[Kochte 10] M. A. Kochte, M. Schaal, H.-J. Wunderlich, and C. G. Zoellin. “Efficient fault simulation on many-core processors”. *Proceedings IEEE Design Automation Conference*, 2010:380–385.

[Lee 06] H. Lee, S. Natarajan, S. Patil, and I. Pomeranz. “Selecting high-quality delay tests for manufacturing test and debug”. *Proceedings IEEE International Defect and Fault Tolerance in VLSI Systems Symposium*, 2006:59–70.

[Lee 02] S. Lee, B. Cobb, J. Dworak, M. Grimaila, and M. Mercer. “A new atpg algorithm to limit test set size and achieve multiple detections of all faults”. *Proceedings IEEE Design, Automation and Test in Europe*, 2002:94.

[Li 10] M. Li and M. S. Hsiao. “Fsimgpˆ 2: An efficient fault simulator with gpgpu”. *Proceedings IEEE Asian Test Symposium*, 2010:15–20.

[Liao 11] K.-Y. Liao, C.-Y. Chang, and J.-M. Li. “A parallel test pattern generation algorithm to meet multiple quality objectives”. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(11), 2011:1767–1772.

[Liao 13] K.-Y. Liao, S.-C. Hsu, and J. C.-M. Li. “Gpu-based n-detect transition fault atpg”. *Proceedings IEEE Design Automation Conference*, 2013:28.

[Liao 14] K.-Y. Liao, A.-F. Lin, J. C.-M. Li, m. S. Hsiao, and L.-T. Wang. "Gpu-based timing-aware test generation for small delay defects". *Poster of IEEE European Test Symposium*, 2014.

[Lin 06] X. Lin, K.-H. Tsai, C. Wang, M. Kassab, J. Rajski, T. Kobayashi, R. Klingenberg, Y. Sato, S. Hamada, and T. Aikyo. "Timing-aware atpg for high quality at-speed testing of small delay defects". *Proceedings IEEE Asign Test Symposium*, 2006:139–146.

[Lin 08] Y.-T. Lin, O. Poku, N. Bhatti, and R. Blanton. "Physically-aware n-detect test pattern selection". *Proceedings IEEE Design, Automation and Test in Europe*, 2008:634–639.

[Liu 12] Y. Liu and Q. Xu. "On modeling faults in finfet logic circuits". *Proceedings IEEE Internaional Test Conference*, 2012:1–9.

[Ma 95] S. Ma, P. Franco, and E. McCluskey. "An experimental chip to evaluate test techniques: Experimental results". *Proceedings IEEE International Test Conference*, 1995:676–683.

[Malaiya 82] Y. K. Malaiya and S. Y. H. Su. "A new fault model and testing technique for cmos devices". *Proceedings IEEE International Test Conference*, 1982:25–34.

[Maxwell 92] P. Maxwell, R. Aitken, V. Johansen, and I. Chiang. "The effectiveness of iddq, functional, and scan tests: How many fault coverages do we need?" *Proceedings IEEE International Test Conference*, 1992:676–683.

[Maxwell 91] P. C. Maxwell, R. C. Aitken, V. Johansen, and I. Chiang. "The effect of different test sets on quality level prediction: When is 80% better than 90%?" *Proceedings IEEE International Test Conference*, 1991:358.

[McCluskey 00] E. J. McCluskey and C.-W. Tseng. "Stuck-fault tests vs. actual defects". *Proceedings IEEE International Test Conference*, 2000:336–342.

[Mei 74] K. C. Y. Mei. "Bridging and stuck-at faults". *IEEE Transactions on Computers*, 1974:720–727.

[Mitra 04] S. Mitra, S. Eichenberger, E. Volkerink, and E. J. McCluskey. "Delay defect screening using process monitor structures". *Proceedings IEEE VLSI Test Symposium*, 2004:43–43.

[Meuhldorf 76] E. Muehldorf. "Designing lsi logic for testability". *Proceedings IEEE International Test Conference*, 1976:45–49.

[Nigh 97] P. Nigh, W. Needham, K. Butler, P. Maxwell, and R. Aitken. "An experimental study comparing the relative effectiveness of functional, scan, iddq and delay-fault testing". *Proceedings IEEE VLSI Test Symposium*, 1997:459–464.

[Patil 90] S. Patil and P. Banerjee. "A parallel branch and bound algorithm for test generation". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(3), 1990:313–322.

[Peng 10] K. Peng, J. Thibodeau, M. Yilmaz, K. Chakrabarty, and M. Tehranipoor. "A novel hybrid method for sdd pattern grading and selection". *Proceedings IEEE VLSI Test Symposium*, 2010:45–50.

[Qiu 04] W. Qiu, L.-C. Wang, D. Walker, D. Reddy, X. Lu, Z. Li, W. Shi, and H. Balachandran. "K longest paths per gate (klpg) test generation for scan-based sequential circuits". *Proceedings IEEE Test Conference*, 2004:223–231.

[Rearick 06] J. Rearick. "A survey of test problems and solutions". *Proceedings IEEE Internaltional Test Conference*, 2006:1–10.

[Roth 76] J. P. Roth. "Method of testing for shorts". *IBM Technical Disclosure Bulletin*, 1976:3108–3109.

[Roth 66] J. P. Roth. "Diagnosis of automata failures: A calculus and a method". *IEEE Transactions on Computers*, 1981:676–683.

[Sato 05] Y. Sato, S. Hamada, T. Maeda, A. Takatori, Y. Nozuyama, and S. Kajihara. "Invisible delay quality-sdqm model lights up what could not be seen". *Proceedings IEEE International Test Conference*, 2005:9–12.

[Schulz 88] M. H. Schulz, E. Trischler, and T. M. Sarfert. "Socrates: A highly efficient automatic test pattern generation system". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 7(1), 1988:126–137.

[Seshu 65] S. Seshu. "On an improved diagnosis program". *IEEE Transactions on Electronic Computers*, (1), 1965:76–79.

[Shao 02] Y. Shao, I. Pomeranz, and S. M. Reddy. "On generating high quality tests for transition faults". *Proceedings IEEE Asign Test Symposium*, 2002:1–8.

[Smith 85] G. Smith. "Model for delay faults based upon paths". *Proceedings IEEE International Test Conference*, 1985:342–349.

[Smith 87] S. P. Smith, B. Underwood, and M. R. Mercer. "An analysis of several approaches to circuit partitioning for parallel logic simulation". *IEEE Transactions on International Conference on Computer Design*, 1987:664–667.

[Venkataraman 04] S. Venkataraman, S. Sivaraj, E. Amyeen, S. Lee, A. Ojha, and R. Guo. "An experimental study of n-detect scan atpg patterns on a processor". *Proceedings IEEE VLSI Test Symposium*, 2004:23–28.

[Wadsack 78] R. L. Wadsack. "Fault modeling and logic simulation of cmos and mos integrated circuits". *Bell System Technical Journal*, 1978:1449–1474.

[Waicukauski 87] J. Waicukauski, E. Lindbloom, B. K. Rosen, and V. Iyengar. "Transition fault simulation". *IEEE Transactions on Design and Test of Computers*, 4(2), 1987:32–38.

[Waicukauski 89] J. A. Waicukauski and E. Lindbloom. "Failure diagnosis of structured vlsi". *IEEE Transactions on Design and Test of Computers*, 6(4), 1989:49–60.

[Yilmaz 08b] M. Yilmaz, K. Chakrabarty, and M. Tehranipoor. "Interconnect-aware and layout-oriented test-pattern selection for small-delay defects". *Proceedings IEEE International Test Conference*, 2008:1–10.

[Yilmaz 08a] M. Yilmaz, K. Chakrabarty, and M. Tehranipoor. "Test-pattern grading and pattern selection for small-delay defects". *Proceedings IEEE VLSI Test Symposium*, 2008:233–239.