EFFICIENT TEST RELAXATION TECHNIQUES FOR COMBINATIONAL CIRCUITS

By

ALI SALEH AL-SUWAIYAN

A Thesis Presented to the

DEANSHIP OF GRADUATE STUDIES

In Partial Fulfillment of the Requirements

for the Degree of

MASTER OF SCIENCE

IN

COMPUTER ENGINEERING

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

Dhahran, Saudi Arabia

October 2002

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS

DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by

ALI SALEH AL-SUWAIYAN

under the direction of his thesis advisor and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER ENGINEERING

Thesis Committee

Dr. Aiman El-Maleh
(Chairman)

Dr. Sadiq Sait
(Member)

Dr. Mostafa Abd-El-Barr
(Member)

Dr. Sadiq Sait
(Department Chairman)

Dr. Osama A. Jannadi
(Dean of Graduate Studies)

Date

To my parents for their support and guidance.
To my dear wife for her love and patience.

Acknowledgement

First of all, all sincere praises and thanks are due to our God, Allah (SWT), who created us and guided us to this great religion, i.e., the Islam. No one can thank God as he deserves because of his limitless blessings on us. May God's peace and blessings be upon his prophet Mohammad who said[1] : *The person who does not thank people, does not thank God as well.* After that comes the following.

Thanks to my advisor, Dr. Aiman El-Maleh, who was very helpful and nice during all the phases of this work. Also, thanks to Dr. Sadiq Sait and Dr. Mostafa Abd-El-Barr, for their valuable comments and constructive criticisms. Thanks are also due to our great department, Computer Engineering, and our great university, KFUPM, for valuable support. Also, I would like to express my deepest thanks to every instructor who contributed in building my knowledge and experience. Thanks are also due to my colleagues, especially Mr. Khalid Al-Utaibi, who did not hesitate in providing me with any kind of help I need.

I also thank my great parents who provided and supported me all the way. Thanks to them and may God bless them. I also thank my brother, Dr. Mohammed, for his valuable advices, comments, and support. Last but not least, thanks to my great dear wife who supported me with love and patience. She was really very patient during my work and she always inspires me to finish my work as quickly as possible. Thanks to her very much.

Finally, thanks to everybody who contributed to this achievement in a direct or an indirect way.

[1]This is only a translation of the meaning of what he said. He originally said so in Arabic. May peace and blessings be upon him.

Contents

List of Tables

List of Figures

List of Algorithms

THESIS ABSTRACT

Name	:	Ali Saleh Mohammed Al-Suwaiyan
Title	:	Efficient Test Relaxation Techniques For Combinational Circuits
Major Field	:	Computer Engineering
Date of Degree	:	October 2002

The significant advancement in VLSI technology has made System-On-Chip (SOC) designs very popular. One of the most challenging problems in testing SOCs is dealing with the large volume of test data. There have been two methods to release this problem, namely, test compaction and test compression. Many compression techniques assume relaxed test set in order to achieve high compression ratios. In this work, we address the problem of generating a relaxed test set from a given test set. A Bitwise Relaxation (BR) technique can be used to solve this problem. However, the BR technique is very slow for large circuits. Another way to obtain a relaxed test set is to generate the test set using dynamic compaction technique. Dynamic compaction is slow as well, and generating the relaxed test set using this method slows down the ATPG process. Furthermore, dynamic compaction can not be used to relax an existing test set. Thus, the only existing solution to the test relaxation problem is the BR method. In this work, we propose three efficient techniques to solve the test set relaxation problem. We also propose cost functions to guide the selection in maximizing the number of extracted x's. The proposed techniques are faster than the BR method by several orders of magnitude. They also obtain comparable results with the BR method.

MASTER OF SCIENCE DEGREE
KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
OCTOBER 2002

Chapter 1

Introduction

System-on-a-chip (SOC) devices are an enabling technology for a wide spectrum of embedded computing applications. The principal characteristics of these devices are that they contain some mixture of processor cores, embedded memory and a variety of mixed signal interfaces, and implement the majority of the functions that previously occupied an entire circuit board onto a single silicon die. The increasing size of SOCs has increased the size of test data significantly. Dealing with large volume of test data is one of the major challenges in testing SOCs [**?**][**?**]. The amount of time required to test SOCs depends on the size of the test data and the speed of the data transfer channel. The cost of testers increases significantly with the increase in their speed, channel capacity, and memory.

The problem of large test storage requirement has been solved by two techniques in the literature, namely test *compaction* and *compression*. The goal of test compaction is to reduce (or compact) the number of test vectors into a smaller number that achieves the same fault coverage. Test compaction can be classified into two classes: *post-generation compaction*

and *compaction during test generation*. In the first class, the number of test vectors is reduced after they are generated, whereas in the second class, the number of test vectors is minimized during the automatic test pattern generation (ATPG) process. Examples of the first class include reverse order fault (ROF) simulation [**?**], forced pair-merging (FPM) [**?**], redundant vector elimination (RVE) [**?**], and essential fault reduction (EFR) [**?**]. Examples of the second class include dynamic compaction [**?**], and COMPACTEST [**?**].

The objective of test compression is to reduce the number of bits needed to represent the test set. For test data compression, it is essential that the compression is lossless. Several test compression techniques have been proposed [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**].

Compaction and compression techniques can achieve better results if the test set is composed of test cubes, i.e., if the test set is partially specified or *relaxed*. In fact, most compression techniques in the literature assume a relaxed test set. Without the dynamic compaction option, ATPGs generally generate fully specified test sets.

Besides compaction and compression, test relaxation can also help in test power reduction as well. The unspecified portions of the test set can be specified in a way to reduce the power dissipation during the scan-in of tests [**?**].

One obvious way to solve the test relaxation problem, i.e., extracting a partially specified test set from a fully-specified one, is to use a Bitwise Relaxation (BR) method, where every bit is tested whether changing it to an x reduces the fault coverage or not. Obviously, this method is $O(nmk)$, where n is the number of primary inputs, m is the number of test vectors, and k is the time needed to fault simulate the circuit under the given test set. Obviously, this method is impractical for large circuits.

A partially specified test set can also be obtained using dynamic compaction technique [**?**].

In dynamic compaction, every partially specified vector is processed immediately after its generation by trying to specify primary inputs (PIs) that are unspecified so that it will detect additional new faults. Generally, in dynamic compaction, the unspecified assignments on primary inputs are filled with random values. This feature can be disabled to obtain a compact and relaxed test set. However, Dynamic compaction slows down the test generation process. In addition, it cannot benefit from random test pattern generation, because this technique is fault-oriented. Furthermore, this technique does not solve the problem of relaxing an already existing test set. Thus, effectively, the only solution to the problem of relaxing a given test set is the BR method. However, a recent and parallel research has been conducted to solve the relaxation problem. This work has been proposed by Kajihara and Miyase [**?**] in 2001. This is discussed in Chapter 2. Due to the unavailability of the experimental test sets used in [**?**], no comparison is made with this technique.

This thesis addresses the test relaxation problem. We propose three solutions to this problem. The first solution is based on the critical path tracing (CRIPT) algorithm. As with CRIPT, this solution is not exact in the sense that the fault coverage might reduce after relaxation. However, as will be shown in the experimental results, the drop in fault coverage is small for most of the benchmark circuits. This solution is referred to as the CRIPT-based relaxation (CRIPTR) technique.

In the second proposed solution, the relaxed test set maintains the same fault coverage of the original test set. This solution does not utilize CRIPT, but it simply identifies all the newly detected faults under a given test vector and marks all the lines whose values are necessary to detect the faults. Unmarked inputs are relaxed. This solution is referred to as the Single-Value Relaxation (SVR) technique. This name has been chosen since this technique justifies only a single value which is the fault-free value. This is unlike the third

solution, which improves the SVR technique by justifying both the fault-free and the faulty machines, without worrying from where a controlling value is justified -as it is the case in the previous two solutions- as long as the considered faults are detected. The third solution is referred to as the Two-Values Relaxation (TVR) technique.

Usually, there are more than one relaxed version of a given fully specified test set, and some versions give more x's than others. Finding the version with the maximum number of x's is an NP-complete problem. However, cost functions are utilized to maximize the number of unspecified bits. In this work, we propose new *fanout-based cost functions*, which, in average, outperform regular cost functions [**?**] with respect to the number of extracted x's.

All the proposed techniques obtain comparable relaxation quality as compared to the BR method, and they are faster by several orders of magnitude. However, as compared to each other, the SVR and the TVR techniques achieve better results in terms of CPU time and fault coverage. In terms of the average relaxation quality, the CRIPTR technique obtains a slightly better average than both the SVR and the TVR techniques.

1.1 Problem Definition

The problem under consideration can be defined as follows:

> *Given a fully specified test set T of a given combinational circuit, it is required to relax T while maintaining the fault coverage and maximizing the percentage of x's.*

This problem is referred to as the *test relaxation problem*.

1.2 Thesis Organization

This thesis is organized as follows. In Chapter 2, some basic background is covered. Then, the CRIPT Algorithm is reviewed. This is followed by a description of a number of compaction and compression techniques and how they can be improved when provided with relaxed test sets. A discussion of the existing solution presented by Kajihara and Miyase in [**?**] is also presented in Chapter 2. In Chapter 3, the three proposed relaxation techniques are presented together with the proposed fanout-based and combined cost functions. Experimental results are given in Chapter 4. Finally, the thesis ends by some conclusions and future directions.

Chapter 2

Literature Review

This chapter provides some necessary background information. The first section lists some concepts and definitions. Next, the critical path tracing algorithm is reviewed. Then, some of the test compaction techniques for combinational circuits are described. This is followed by a number of the existing test compression techniques. Compaction and compression techniques are considered as the motivation of our work since the effectiveness of compression and compaction techniques can generally be improved by relaxing their input test sets. As we will see, relaxation helps a lot in most of the presented compaction techniques. For test compression, relaxation is crucial, because the x's can always be specified to maximize the compression ratio. In this Chapter also, a discussion of the BR method and the work presented by Kajihara and Miyase [**?**] is given.

2.1 Background

The scientific study of any discipline must be built upon rigorous definitions arising from fundamental concepts. What follows is a list of definitions and basic concepts used throughout this thesis. For the sake of clarity, rigor has been sacrificed where appropriate.

2.1.1 Preliminaries

The Single Stuck-Fault Model

VLSI circuits are vulnerable to several kinds of physical faults [**?**]. Logical faults are used to represent the effect of physical faults on the behavior of the modelled system. One advantage of using logical faults to model physical faults is that the problem of fault analysis becomes a logical instead of a physical problem. The complexity of such a logical problem reduces greatly since many physical faults may be modelled by one logical fault. Another advantage is that some logical fault models are technology-independent in the sense that the same fault model is applicable to many technologies. Also, tests derived for logical faults may be used for physical faults whose effects on circuit behavior is not obvious or is too complex to be analyzed.

Throughout this thesis, it is always assumed that the system has at most one logical fault. This simplifying *single-fault assumption* is justified by the frequent testing strategy, in which a system should be tested often enough so that the probability of more than one fault developing between two consecutive testing experiments is sufficiently small.

In this work also, the *Single-Stuck Fault* (SSF) Model is assumed. This fault model belongs to *structural fault models*. Structural fault models assume that components are fault-free and only their interconnections are affected. In the SSF model, a signal line l in the Circuit Under Test (CUT) can be *stuck-at* a logical value v ($v \in \{0, 1\}$), and this is denoted by l *s-a-*v, or l/v. This logical fault model is also refereed to as the *classical* or *standard* fault model because it has been the first and most widely studied and used. Although its validity is not universal, its benefits are apparent from the following characteristics:

- It represents many different physical faults.
- It is technology-independent, as the concept of a single line being stuck at a logic value can be applied to any structural model.
- Experience has shown that tests that detect SSFs detect many nonclassical (eg., bridging) faults as well.
- Compared to other fault models, the number of SSFs in a circuit is small, and the number of faults that should be explicitly analyzed can be reduced by fault-collapsing techniques.
- SSFs can be used to model other types of faults.

Fault Detection

In a given CUT, an SSF could be *detectable* or *undetectable*. In a non-redundant combinational logic circuit, all SSFs are detectable; however, a redundant combinational logic circuit may contain undetectable (or redundant) faults. A *test vector* is a string of bits that

is applied to the primary inputs (PIs) of a combinational circuit to detect one or more SSFs. A *test cube* is a partially specified test vector, where only the necessary PIs for detecting the targeted fault(s) are assigned binary values (0 or 1). The other PIs are left as x's. If all PIs are assigned binary values, then the vector is *fully specified*. Two test cubes are *compatible* if they don't specify contradicting PI assignments. For example, the two test vectors $t_1 = 001x$ and $t_2 = x0x1$ are compatible, whereas the two test vectors $t_3 = 001x$ and $t_4 = 10x1$ are incompatible because they have contradicting assignments in their most significant bits. Given a collection of test cubes, i.e., a *test set*, the *fault coverage* of this collection is defined to be the ratio between the number of faults detected by such test set and the total number of faults under consideration. When dealing with sequential circuits, the order of the test cubes within a given test set matters and should be preserved, otherwise, the fault coverage might get affected. However, for combinational circuits, ordering of test cubes within a test set does not matter and does not have any effect on the fault coverage.

Independent & Compatible Faults

A set of faults, F, is said to be *independent* if no two faults in F can be detected by a single test vector. Clearly, the cardinality of the largest independent fault set for a given CUT is a lower bound on the number of test vectors that can cover a given set of faults. On the other hand, a *compatible fault set* contains faults that are detectable by a single test vector.

Essential Faults

The following definitions assume a given fault list F of a circuit and a given test set T for F.

Definition 2.1: The set of faults detectable by a pattern t in T is denoted by DET(t).

Definition 2.2: The *essential faults* of a pattern t in T, ESS(t), are the faults which are detected by t only and not by any pattern in T other than t.

Definition 2.3: The *potential essential faults* of a set of patterns $P \subseteq T$, PESS(P), are the set of faults that are detected by every vector in P but not by any other vector in $T - P$.

Notice that the last definition implies that every vector in P detects (among other faults, if any) every fault in PESS(P), and every fault in PESS(P) is detected by every vector in P. However, no vector in $T - P$ detects any fault in PESS(P), and no fault in PESS(P) is detected by any pattern in $T - P$.

ATPGs & RPGs

An Automatic Test Pattern Generator (ATPG) is an algorithm that generates a test set to detect given target faults. Examples of ATPG algorithms include D-Algorithm, PODEM, FAN, TOPS, FAST [**?**]. These ATPGs are referred to as deterministic ATPGs. On the other hand, there are Random Pattern Generators (RPGs), which involve only generation of random test vectors without targeting certain faults.

Full-Scan Design

To simplify testing of sequential circuits, flip-flops are modified to become directly controllable from primary inputs, and observable through primary outputs. A scan register or a scan chain is a set of such modified flip-flops that have a serial-in pin that is used to shift the

desired values into the register and a serial-out pin to shift the content of the register out. In *full-scan* design, the scan chain is composed of all flip-flops of the circuit. In *partial scan* design, only a partial set of flip-flops is included in the scan chain. The proposed techniques works only for full-scan (or combinational) circuits.

Definitions Related to the Proposed Work

Definition 2.4: *Relaxation Quality* of a given partially specified test set (denoted by $\%x$) is defined to be the ratio between the number of x's (unspecified bits) and the total number of bits in the test set. Mathematically speaking,

$$\%x = \frac{\text{number of } x\text{'s in the test set}}{\text{total number of bits in the test set}}$$

Throughout this thesis, the terms *relaxation quality*, and *percentage of x's* are used interchangeably.

Definition 2.5: Let line l be an input of a gate G. The *output of* l is defined to be the output of G.

Definition 2.6: Let line l be an input of a gate G. The *side inputs* of line l are defined to be the input lines of G other than l.

Definition 2.7: The *requirement list* L of a given fault is the set of lines whose values are required to detect (i.e., excite and propagate) that given fault.

Definition 2.8: A line l is said to be *reachable* from a stem s if the fault effect in stem s reaches line l.

2.1.2 Critical Path Tracing (CRIPT)

In this section, the main points of the CRIPT algorithm are summarized. This algorithm is presented in [**?**], and is used to fault simulate combinational circuits. Before introducing the algorithm, two necessary definitions are given.

Definition 2.9: A line l has a *critical value* v under the test vector t iff t detects the fault l stuck-at-$\overline{v}$. A line with a critical value in t is said to be critical in t [**?**].

Definition 2.10: A gate input is said to be *sensitive* in a test vector t if complementing its value changes the value of the gate output to a known logic value [**?**].

The following theorem tells us how to determine critical inputs of a gate whose output is critical.

Theorem 2.1:if a gate output is critical, then its sensitive inputs, if any, are also critical [**?**].

Algorithm **??** outlines the main CRIPT algorithm for evaluating a given test vector. It assumes that true-value simulation, including the marking of sensitive gate inputs, has been performed. The algorithm processes every logic cone starting at the circuit primary outputs and alternates between two main operations: critical path tracing inside a fanout-free region (FFR), represented by the procedure $Extend()$, and checking a stem for criticality done by the function $Critical()$. Once a stem j is found to be critical, the algorithm continues tracing from j.

Algorithm **??** shows the recursive procedure $Extend()$, which backtraces all critical paths inside an FFR starting from a given line l by following lines marked as sensitive. $Extend()$ stops at FFR inputs and collects all the stems reached in the set $StemsToCheck$.

Algorithm 2.1 Critical path tracing for one test [**?**].

for every primary output z **do**
  $StemsToCheck \leftarrow \phi$
  $Extend(z)$
  while $StemsToCheck \neq \phi$ **do**
    $j \leftarrow$ the highest level stem in $StemsToCheck$
    remove j from $StemsToCheck$
    if $Critical(j)$ **then**
      $Extend(j)$
    end if
  end while
end for

Algorithm 2.2 $Extend(i)$ [**?**]

```
1:  mark i as critical
2:  if i is a fanout branch then
3:      add the stem of i to StemsToCheck
4:  else
5:      for every input j of i do
6:          if sensitive(j) then
7:              Extend(j)
8:          end if
9:      end for
10: end if
```

Our implementation of the function $Critical()$ is shown in Algorithm **??**. This is different from the two variations proposed in [**?**]. One of the variations in [**?**] has been implemented, however no improvement in terms of CPU time has been observed. The other variation is complex to implement, and due to the unavailability of the CRIPT source code, the implementation shown in Algorithm **??** has been adopted. Although our implementation is less efficient, it is employed for its simplicity. When a stem is non-reconvergent, it is critical. Otherwise, the stem is checked if it has been fault simulated. If not, then fault simulate the stem and check whether the fault propagates to the required output. If it propagates, the stem is critical; otherwise it is not. Reconvergence information are gathered during preprocessing and are independent of the test vector applied.

Algorithm 2.3 $Critical(s)$

```
1: if s is non-reconvergent then
2:     return TRUE
3: else if stem has not been fault simulated then
4:     fault simulate the stem
5: end if
6: if the stem fault propagates to the required output then
7:     return TRUE
8: else
9:     return FALSE
10: end if
```

Sometimes critical path tracing may not identify all the faults detected by a test. This may occur in a test t that propagates the effect of a fault on multiple paths that reconverge at a gate without sensitive inputs in t. More precisely, critical path tracing does not detect faults that can *only* be detected by multiple-path sensitization with reconvergence at a gate with nonsensitive inputs. In Figure **??**, the fault effects of $B/0$ under the test 111 propagate on two paths and reach the reconvergence gate on nonsensitive inputs. In this case, critical path tracing would not reach B and thus would not flag $B/0$ as detected. However, this situation

Figure 2.1: A circuit that shows the approximate nature of the CRIPT algorithm.

occurs seldom in practical circuits, and even when it occurs, the impact is negligible. This is why the critical path tracing algorithm is considered as an approximate algorithm.

2.2 Benefits Of Test Relaxation

This Section describes some test compaction techniques, test compression techniques, and a test power reduction technique. Along with the description, it is mentioned (either implicitly or explicitly) how each technique can benefit from test relaxation. Thus, this section can be considered as our motivation.

2.2.1 Test Compaction Techniques

Test generation for digital circuits has been extensively investigated. Most of these efforts concentrate on how to efficiently generate a complete test set for the given circuit without specifically considering the size of the test set. The test set size, however, is directly proportional to the cost and time of test application, which may involve more than thousands of chips. This test efficiency problem is especially acute in the scan-designed circuits, for which each test pattern must be scanned in before being actually applied. Clearly, a compact test set is highly desirable for economical testing. Furthermore, a compact test set also results in a smaller demand for buffers in Automatic Test Equipments (ATEs) as well as less hardware overhead in an IC with in-chip test storage.

Although the ideal goal of test compaction is to obtain the minimal test set, its computation is unfortunately not feasible because the complexity of calculating the minimum number of tests required to detect all single stuck-at faults in an irredundant combinational circuit has already been proven to be NP-hard [**?**]. Various heuristic methods for test set compaction have thus been proposed [**?**, **?**, **?**, **?**, **?**, **?**, **?**]. These methods can be classified into *compaction during test generation* and *post-generation compaction*. The key idea of compaction during test generation is that if the fault coverage of each test pattern is maximized during test generation, then the total number of patterns can be reduced. In post-generation compaction, a given test set is used as the starting point to perform compaction.

Some compaction techniques can greatly benefit from relaxing their input test sets. In what follows, we describe some of the existing compaction techniques, and whenever possible, we describe how they can be improved when provided with partially specified input test sets.

Compaction During Test Generation

Dynamic Compaction

In 1979, Goel and Rosales [**?**] presented the first dynamic compaction technique. After the generation of a test vector for a given target fault, unspecified bits of the vector are processed by trying to assign them a value so that the vector will detect additional new faults. Algorithm **??** outlines such a technique that is activated after a test t has been successfully generated for the selected target fault. The function $promising$ decides whether t is a good candidate for dynamic compaction. For example, if the percentage of PIs with x values is

less than a user defined limit, t is not processed for compaction. While t is promising, try to extend t (by changing some of the unspecified PIs) such that it detects a selected secondary target fault g. This is done by attempting to generate a test for g using only the unspecified PIs.

Notice that after an ATPG system is used to generate a test vector for the target fault, unspecified bits of the vector can be filled randomly. Then, the vector can be relaxed according to one of our proposed relaxation techniques. The unspecified bits of the vector can be used to detect additional new faults, in addition to the target fault and the faults detected randomly.

Algorithm 2.4 Dynamic Compaction [**?**]

```
while promising(t) do
   Select a secondary target fault g
   Try to extend t to detect g as well
end while
```

Compaction Based On Independent Fault Sets

Akers *et al.* [**?**] provide an algorithm to generate maximal independent fault sets based on the ideas of fault dominance and fault equivalence. The independent fault sets generated form the basis for generating test patterns. From these independent fault sets, a fault matching Algorithm is used to generate sets of compatible faults. The ultimate goal is to generate a minimal number of compatible fault sets, since each compatible fault set is detectable by at least a single test vector. Thus, minimizing the number of compatible fault sets implies minimizing the number of test vectors needed to cover a given set of faults. This technique has an optimistic bound on the size of the largest independent fault set. Tromp [**?**] improved over Akers *et al.* technique in three ways. The independent fault set generation

procedure is improved so that larger independent fault sets and more realistic bounds can be obtained. A heuristic Algorithm is used for this purpose. Another improvement is that faults are matched into compatible fault sets using a bipartite matching algorithm one fault at a time. The implication procedure is also improved to enable more success in finding all compatible fault sets for a single test vector. The results are provided for small benchmark circuits only.

COMPACTEST

COMPACTEST is a system introduced by Pomeranz *et al.* [**?**]. This system starts with a preprocessing step to order the faults in the fault list. The ordering is based on a heuristic which tries to compute *Maximum Independent Fault Set* (MIFS) inside every maximal fanout free region (FFR) of the circuit. The largest independent fault set found is placed on the top of the fault list followed by the next largest and so on. A target fault is taken from the ordered fault list and a test vector is generated for it. This fault is referred to as the *primary target fault*. During the test generation of the primary target fault, preprocessing information is used to obtain a test that detects additional faults in the FFR to which the primary fault belongs. Next comes the main heuristic of COMPACTEST, i.e., *maximal compaction*. The heuristic works as follows. Each specified input of the test vector is considered separately. If the generated test vector is still a test for the fault when the input value is flipped, then the input value is flagged. After all specified inputs are examined, all flagged inputs are turned into x's. It is possible that the resulting vector is no longer a test for the primary target fault; however, this provides maximum flexibility in detecting additional faults by the same test vector. Notice that maximal compaction is similar to the BR technique with only a minor modification. Notice that our proposed techniques can fit

here very well. After a test is generated for the primary target fault, unspecified inputs can be assigned randomly to detect additional faults. Then, one of our proposed techniques can be used to relax unnecessary assignments. Maximal compaction can be applied next.

After the test is maximally compacted, another target fault, which is refereed to as *secondary target fault*, is picked from the ordered fault list and the unspecified bits are used to generate a test for this fault, i.e., the specified inputs are not changed. Whenever a secondary fault detection contradicts with the already specified bits, another secondary target fault is picked and the bits updated to detect the previous secondary target fault are unspecified again. Next, the test generator tries again to detect other faults belonging to the same FFR of the secondary fault. Finally, the part of the test that was specified to detect the secondary target fault is maximally compacted. This process is repeated until all inputs are specified or all the faults in the fault list are tried as secondary target faults. At last, if there are any unspecified bits, they are randomly specified. The test vector obtained is fault simulated and all detected faults are removed from the fault list. Then a new primary target fault is picked from the top of the fault list and the whole process is repeated. The entire process is repeated until either the fault list is empty or all the faults in the fault list are tried as primary faults.

Fault selection and maximal compaction are not the only heuristics employed in COMPACTEST. There is the *rotating backtrace* heuristic. This aims at sensitizing different paths every time a value on a line is to be justified. As a consequence, different faults, which propagate through different paths, are potentially detected by the test vector generated. In this heuristic, each gate is associated with a counter, which is initially 0. Whenever the justification procedure reaches the output of a gate which can be justified by setting only one of its inputs to a controlling value, the procedure selects the input whose number is given by the counter. The counter is then incremented modulo the number of the inputs

of the justified gate. Experiments show that COMPACTEST outperformed all pre-existing dynamic compaction techniques.

Kajihara *et al.* [**?**] improved upon [**?**]. They improved the procedure of computing independent fault sets which are used to pick targets for pattern generation. They also calculated tighter lower bounds for the size of the smallest test set. They improved upon rotating backtrace as well. They also proposed a new compaction technique termed *double detection*. The ultimate goal of double detection is to use input values that remain unspecified during the test generation process in order to increase the potential for obtaining and then dropping redundant test vectors. This is achieved by detecting faults twice before dropping them from the fault list.

COMPACT

Ayari and Karminska [**?**] presented a simple to implement approach for dynamic test vector compaction called COMPACT. After test pattern t_i is generated, instead of selecting the next untested fault from the fault list and attempting test generation, COMPACT is used to compare previously generated test vectors. If t_i is compatible with any previously generated test t_j ($j < i$), then t_j is replaced by $t_j \cap t_i$ and t_i is discarded. The intersection is performed bit-by-bit and an efficient data structure implementation is proposed. Although the experimental results are only a bit inferior to those of [**?**], the proposed procedure itself is considerably more efficient. This technique can benefit a lot from our work as follows. After an ATPG generates a test vector, it can randomly fill unspecified bits to detect more faults. Then, the generated test vector is relaxed and merged with the previously generated test vectors.

Post-Generation Compaction Techniques

Reverse Order Fault Simulation (ROFS)

Schulz *et al.* [**?**] presented an efficient test pattern generator called SOCRATES. It was not their primary concern to have a compact test set. However, they suggested a technique called *reverse order fault simulation* to compact the set of test vectors which forms the output of their system. This compaction technique works as follows. A fault simulation in reverse order is carried out at the final stage of the ATG process. In other words, the test set is fault simulated in reverse order of generation. Any test vector that does not detect a fault undetected by the vectors simulated earlier is dropped from the test set, and hopefully we end up with a compact version of the generated test set. It is observed experimentally that often all the covered faults are detected using only a subset of the original set of test vectors. The intuitive reason behind this is simply that the test vectors which are further down the list detect faults which are most difficult to detect. Therefore, if we fault simulate a test which is at the end of the list first, it not only detect a difficult fault, it also detects many other faults by pure chance. This way, difficult faults are removed away early. It is a fairly effective technique for a first compaction attempt. This technique also indicates the importance of the way in which faults are ordered in the fault list.

ROFS technique may benefit from our proposed work as follows. After a test set is compacted according to this technique, relax it. Then, fill the unspecified bits randomly. Then apply ROFS again, in a hope that some of the vectors became redundant after the random filling of the relaxed test set. This process is repeated until the test set size stabilizes.

Reverse Order Test Compaction (ROTCO)

Reddy *et al.* introduced ROTCO [**?**] method which is similar to ROFS method with one difference: it allows test vectors to be changed during the compaction process. This increases the flexibility in detecting faults detected by earlier vectors and results in a more compacted test set. Although the original test set is modified in this method, the original fault coverage is not compromised.

To perform reverse order test compaction, we need to keep track of two things: (1) The faults detected by each vector, and not detected by any test vector generated earlier, and (2) The positions of randomly specified bits that are filled by the ATG system after the generation of a test vector. For (2), a bit filled randomly with 0(1) is denoted by $x_o(x_1)$. This information is readily available in most ATG systems. However, if it is desired to compact a randomly generated test set using this technique, then our proposed techniques can help in relaxing the test set before applying ROTCO on it. However, the difference here is that ROTCO does not need to know whether an unspecified bit was a zero or a one, because unspecified bits generated by our techniques do not contribute to the fault coverage of the whole test set.

Let the test vectors to be compacted be $t_1, t_2, \cdots, t_k$. Let t_i detect the set of faults F_i, where F_i contains all the faults detected for the first time by t_i. Consider the tests in reverse order, i.e., $t_k, \cdots, t_2, t_1$. When t_i is considered, all x_o and x_1 values in t_i are first changed into x (a conventional don't-care). This is done to have maximum flexibility in the compaction process. The fault lists $F_1, F_2, \cdots, F_{i-1}$ are considered in the order of increasing cardinality. Let $|F_{j_1}| \leq |F_{j_2}| \leq \cdots \leq |F_{j_{i-1}}|$, where the index $j_n \in \{1, 2, \cdots, i-1\}$ and $n \in \{1, 2, \cdots, i-1\}$. An attempt is made to detect every fault in $F_{j_1}, F_{j_2}, \cdots, F_{j_{i-1}}$ (in this order), by specifying only x values in t_i which have not yet been specified. The order

within a set F_{j_k} is arbitrary. COMPACTEST [**?**] is employed for this purpose. As some lines in the circuit are already set to specified values, the test generation process is simpler than test generation that starts with all inputs unspecified. At the end, if an input is still at x, it is changed back to its original value.

The modified test t_i, resulting from the above process, may differ from the original t_i, so it is possible that it no longer detects some of the faults in F_i that were detected due to random specification of values. Therefore, fault simulation is carried out for the faults in F_i. If any fault is not detected by the modified t_i, then t_i is restored to its original form. This check ensures that the fault coverage is kept unchanged. In test generation procedures aimed at small test sets, like COMPACTEST, a test vector typically detects a large number of faults, and as a result, when the unspecified inputs are specified randomly at the end, hardly any additional (new) faults are detected. Hence, when some of the randomly set values in t_i are changed, the modified t_i still detects all the faults in F_i in most cases.

If the modified t_i passes the above check, it replaces the original t_i in the test set. Fault simulation is performed under t_i for all the faults in $F_1, F_2, \cdots, F_{i-i}$. All the detected faults are removed from their respective fault lists and are added to the fault list F_i. If some fault list $F_j, 1 \leq j < i$, becomes empty, t_j is dropped from the test set.

The above process is repeated for all $t_i, 1 \leq i \leq k$. The vectors left in the test set at the end of the procedure form the compacted test set for the circuit under consideration.

The reason behind ordering the fault lists in increasing order of cardinality is that all the faults in F_j are needed to be detected before the corresponding vector t_j can be dropped from the test set. Targeting the vector with the fewest associated faults first may lead to elimination of more vectors.

Experimental results on ISCAS85 and PLA benchmark circuits show that ROTCO performs significantly better than ROFS, even for highly compacted test sets.

Compatible Pairs Merging

Two compatible test vectors can be combined (compacted) into one test vector by intersecting the two vectors. Pairs merging [**?**] works by merging compatible test vectors with each other. For example, the test vector $0x1$ can be merged with the test vector $x11$ because the two vectors don't specify any contradicting assignment. The resulting test vector is 011 and is considered to represent both of them. Clearly, if the first vector was 001, the two vectors are no longer compatible, and thus cannot be compacted. The quality of this compaction technique heavily depends on the percentage of unspecified bits in the test vectors to be merged. The more unspecified bits, the higher chance for a pair of vectors to be compatible, and thus be merge able.

Forced Pair-Merging (FPM) Algorithm

The FPM algorithm [**?**] is based on the observation that if one vector t in T is replaced by another vector t' to produce a new test set T', then T' has at least the same fault coverage of T provided that ESS(t) $\subseteq$ DET(t').

In the FPM algorithm, raising a bit of a pattern is the basic operation. Raising the bth bit of test t means trying to set the bth bit of t to x while preserving the coverage of ESS(t). If the operation fails, the bit is restored to its original status. Although this operation can be done with the help of a fault simulator, the authors employed a modified logic simulator to perform bit raising operation.

The FPM algorithm works as follows. It takes the first pattern from T as a seed pattern, say t_1, and raises t_1 as far as possible, so as to have the most x's while being still able to detect ESS(t_1). Then for each of the remaining patterns, say t_2, it tries to raise those bits of t_2 which are incompatible with the raised t_1. Note that this step can be accomplished by using one of our techniques by giving low weights for incompatible bits and high weights for the remaining bits and relaxing t_2 based on these weights. Regardless how this process is done, if it succeeds, the pair is merged into a new pattern which replaces t_1 as the seed pattern. T is then updated by removing t_2. Otherwise, the raised t_1 remains intact, and another pattern is selected to be merged with t_1. After trying all the patterns in T, the resultant t_1 is removed from T and stored in the originally empty T^*. This process is repeated until T becomes empty. The resultant T^* is the compacted test set with fault coverage no less than the original one.

In what follows, a description of the bit raising operation is given. The authors have observed that when raising a pattern, the value of each line in the circuit can only be changed from specified to unspecified, and moreover, a fault can only become an undetected one from a detected one. Based on this observation, a modified logic simulator is developed to monitor the detectability of the faults in ESS(t), given that pattern t is to be raised. Monitoring the detectability of ESS(t) means making sure that every fault in ESS(t) is excited and propagated to at least one primary output. This is accomplished by making sure that when a bit is raised, none of the lines which belong to any of the sensitization paths of the faults in ESS(t) or their side inputs has become an x. If any of these lines becomes an x, then at least one fault in ESS(t) is masked through at least one primary output, and the raising fails. However, the bit raising Algorithm takes care of the faults that are detected through multiple outputs. After t is raised, there might be some of the faults in ESS(t) that are undetected due

to fault masking. So fault simulation is performed, and if any of the faults is undetected, the Algorithm will raise the pattern by the help of a fault simulator. Note that raising a pattern t is nothing but relaxing t, and that this technique is heavily dependent on test relaxation. So, instead of the bit raising proposed by the authors, our proposed techniques can be utilized here very well.

Essential Fault Pruning (EFP)

The EFP algorithm [**?**] is a generalization of the FPM algorithm in the sense that for a given pattern t, EFP tries to actively modify the rest of the test set to detect the essential faults of t. If this succeeds, then t can be removed from the test set. Since ESS(t) is now to be detected not only by a single vector but by the whole remaining modified test set, there are evidently more chances to succeed in further reduction of the test set.

In EFP, a pattern t can be removed if ESS(t) can be detected by modifying other patterns so that they detect ESS(t). An essential fault of t is said to be pruned, if it becomes detectable by another pattern after modification. If all the faults in ESS(t) are pruned, then t can be removed from the test set.

Modification of another pattern t_1 to further detect an additional fault f in ESS(t) is done by generating another pattern t_2 such that (DET(t_1) $\cup f$) $\subseteq$ DET(t_2). A procedure called Multiple Target Faults Test Generation (MFTG) is given in the paper and is used to check the existence of such a pattern. For a set of target faults S, MFTG finds a test pattern that detects all the faults in S. If any fault in ESS(t) can not be covered by other patterns, EFP returns fail, and the original test set must be restored. Clearly, the quality of compaction depends on the starting vector t.

The Two By One (TBO) Compaction Algorithm

This Algorithm is proposed by Kajihara *et al.* [**?**]. It is based on the addition of one test vector to replace two other test vectors in a given test set. For a given test set T, the Algorithm works as follows. Essential faults for every test vector are collected, and, as an optional step, maximal independent fault set F is determined. Note that if F is not computed, it is simply set to ϕ. Then, two test vectors t_i and t_j are selected, such that at least one of them, say t_i, does not have essential faults from F. It is also required that the number of essential faults in each vector does not exceed a predetermined limit L, so that things do not get complicated. L is set to twice the average number of essential faults per test vector, but not larger than 50.

Even if none of the faults in ESS(t_i) belongs to F, a pair of independent faults might exist in ESS(t_i) $\cap$ ESS(t_j). So, it is worth to check that no such pair exists before starting test generation.

A procedure for generating a test vector with more than one target fault which is required to detect faults in ESS(t_i) $\cap$ ESS(t_j), is given. This procedure utilizes another procedure proposed by Rajski and Cox in [**?**] which computes the union of the necessary assignments of all target faults. If any conflict is found, the procedure terminates indicating that no test vector can be found. Then, values for detecting each fault are determined using dynamic compaction techniques including the maximal compaction technique [**?**]. Since maximal compaction is applied, the procedure must make sure that the final test vector generated detects all the target faults.

Every additional test vector generated is placed in a set T'. At the end of the process, elements of T' are inserted into T. Redundant test vector are removed by fault simulating T

in the following order. The added test vectors are simulated first. Test vectors for which no added tests are generated are then simulated. Test vectors expected to be removed are then simulated. Any test vector that detect only previously detected faults is removed.

Clearly, if the test set is partially specified, the chances of success in this technique increase. Thus, our proposed techniques can be utilized to relax the test set before applying them on this technique.

Experiments on ISCAS85 and full-scan versions of ISCAS89 are provided. Results show that the sizes of the compacted test sets are close to the calculated lower bound for most of the circuits.

A Set Cover Model For Optimal Test Compaction

Hochbaum [**?**] proposed an optimal test compaction algorithm for combinational circuit that is based on the modelling of test compaction problem as a set cover problem as follows. In the set cover problem, the goal is to identify the smallest collection of sets that cover a given set of elements. In the compaction problem, the elements are the faults to be covered by the given collection of test vectors. Suppose that the circuit contains detected faults named $f_1, \cdots, f_m$ and the test set to be compacted is composed of the test vectors $v_1, \cdots, v_n$. Each test vector v_j is associated with a subset of faults, S_j, that are detectable by v_j. The problem is to find the smallest sub-collection of vectors in which their associated subsets of faults cover the faults $\{f_1, \cdots, f_m\}$. An $m \times n$ detection matrix D is generated. The entry d_{ij} equals one if fault i is detected by test vector j, otherwise it is zero. The author formulates

the set cover problem as an integer programming problem as follows:

$$\begin{aligned} \min \quad & \textstyle\sum_j x_j \\ & Dx \geq e \\ & x_j \in \{0,1\} \quad j \in \{1,\cdots,n\} \end{aligned}$$

Minimizing the sum means minimizing the number of test vectors that will cover the given fault list. The author solves the problem by an integer programming technique called Linear Programming Relaxation (LPR). LPR removes the requirements that the variables are integer, resulting in the new constraints that $x_j \in [0,1]$ for $j \in \{1,\cdots,n\}$. An easy rounding heuristic is then applied if the resulting solution is not integer: if the optimal solution of the LPR is x^*, then the output of the rounding heuristic is $\lceil x^* \rceil$ and thus obviously feasible. Experimental results are given for a subset of ISCAS85 benchmark combinational circuits. As shown in the experimental results, for test sets compacted using COMPACTEST [**?**], this technique only achieves little further compaction for most of the circuits. The author declares that the running time of the LPR procedure is low enough to incorporate it as a final step to follow any test generation procedure.

Redundant Vector Elimination (RVE)

In 1998, Hamzaoglu and Patel in [**?**] have presented two compaction techniques: *redundant vector elimination (RVE)* and *essential fault reduction (EFR)*. These two techniques and the dynamic compaction algorithm proposed in [**?**] are incorporated into an advanced ATPG

system for combinational circuits termed as MinTest. MinTest generated smaller test sets than the previously published results for the ISCAS85 and full scan versions of ISCAS89.

We discuss here the RVE algorithm, and the EFR is discussed next. In practice and during ATPG process, earlier vectors have the potential to become redundant because of the fact that the faults detected by earlier vectors are also detected by later vectors. RVE algorithm identifies such redundant vectors during test generation and dynamically drops them from the test set as follows. It fault simulates all the testable faults in the fault list and keeps track of the faults detected by every vector, the number of essential faults associated with every vector and the number of times a fault is detected. During the test generation, if the number of essential faults of a vector reduces to zero, the vector becomes redundant and thus is dropped from the test set.

RVE can reduce the size of a test set more than ROFS. This is due to the fact that ROFS can not identify a redundant test vector if some of the faults detected by it are only detected by the test vectors generated earlier. ROFS can only identify a redundant vector, if all the faults detected by it are detected by the test vectors generated later. However, RVE can identify a redundant vector if it detects faults which are detected by other vectors, no matter where these vectors are.

Essential Fault Reduction (EFR)

As mentioned above, this technique is proposed by Hamzaoglu and Patel [**?**]. As a reminder, an essential fault ef_i of a test vector t_i is said to be pruned if a test vector $t_j \neq t_i$ in the test set is replaced by a new test vector t_j^a which detects ef_i, ESS(t_j) and the faults that are detected only by t_i and t_j. Notice that having a partially specified test set helps in finding

t_j^a. Since pruning an essential fault decreases the number of essential faults by one, then if all essential faults of a test vector are pruned, the test vector becomes redundant and it can be removed from the test set. After the initial test set is generated (by MinTest), EFR algorithm is used iteratively to further compact the test set by pruning the essential faults of each vector as much as possible. A multiple target test generation (MTTG) [**?**] procedure is used to generate a test vector that will detect a given set of target faults. EFR is an improvement over both the TBO algorithm [**?**] and the EFP algorithm [**?**].

2.2.2 Test Compression Techniques

The main objective of test set compression is to transform the test set into another encoding domain provided that the transform is invertible. In most cases, the number of bits needed to encode the compressed form of the test set is considerably smaller than the number of bits in the original test set, and this is the main objective of test compression. One aspect to compare test compression schemes is the *compression ratio*. Compression ratio, r, gives a measure of the amount of data reduced after the compression process. One popular way to compute the compression ratio is:

$$r = \frac{S_o - S_c}{S_o} \times 100\%$$

Where, S_o is the size of the original test set, and S_c is the size of the compressed test set, both in bits.

A compression technique is said to be successful if it has a high average compression ratio and simple and efficient decompression process.

It is clearly observed that, for any compression scheme, having a partially specified test set helps in increasing the compression ratio. The reason is simple; given a compression Algorithm, then the compression ratio is a function of the data to be compressed, and if there is some freedom in specifying the contents of the data (e.g., some bits are unspecified), one could specify the unspecified portions of the data such that the compression ratio is maximized. Thus, having a partially specified test set is crucial for all test compression techniques.

Let's have a look at some general purpose compression algorithms. A simple compression scheme is *run-length coding*, where a sequence of equal symbols is encoded into two elements, the repeating element and the length of the sequence. For example, run-length coding of $xx1110xx$ is $(x, 2)$, $(1, 3)$, $(0, 1)$, and $(x, 2)$. Note that the parentheses and commas are only for readability, and they don't appear in the compressed data. For data with many long sequences of equal symbols, run-length coding is apparently efficient. Huffman coding is more sophisticated than run-length coding and yields better compression [**?**]. Huffman coding builds a binary tree based on the probability of the occurrence of the letters, where leaves in the binary tree correspond to the letters. The Huffman code for a letter is obtained by traversing the tree from its root node to the leaf corresponding to the letter, concatenating "0" to the code word every time it traverses over a left branch and "1" over a right branch. A different approach from the above two methods is arithmetic coding [**?**]. Arithmetic coding generates a unique tag or identifier for a given sequence of symbols; then deciphers tags to restore the original sequence. The main advantage of arithmetic coding over Huffman coding is that building a binary tree structure is unnecessary. The Lempel-Ziv (LZ) method, based on the construction of a dictionary, builds a list of patterns. Then, patterns are encoded according to their indices in the list [**?**]. The LZ method doesn't re-

quire a priori knowledge of the probability of the occurrence of letters, and becomes more efficient for a longer sequence of patterns whose characteristic is static. The Lempel-Ziv-Welch (LZW) algorithm, which is a derivative nf the LZ method, collects new phrases into a dictionary [**?**]. When a repeating phrase is found, the index of the phrase in the dictionary is recorded to compress the phrase. Some compression utilities available on personal computers and workstations implement variations of the LZW method. The Lempel-Ziv-Storer-Szymanski (LZSS) algorithm keeps track of the last n bytes of data [**?**]. When a phrase appears before being encountered, the phrase is encoded as a pair of values corresponding to the position of the phrase in the buffer and the length of the phrase. Besides the above general data compression algorithms, there are many other compression methods designed for special applications such as speech, image, and video. In what follows, we discuss some of the techniques aimed at compressing test data.

Serial Scan Test Vector Compression

Su and Hwang [**?**] presented a serial scan test vector compression methodology for the test time reduction in a scan-based test environment. This is achieved by carefully examining the relationship between two consecutive test vectors. If the first few bits of the second vector are the same as the last few bits of the first vector, then the matching bits need not be shifted in again, because they are already in the scan chain. As an example, consider the two consecutive vectors 10111011 and 11101011. The last four bits of the first vector (1011) matches the first four bits of the second vector (1011). Thus, instead of shifting in all the 16 bits, we only need to shift in 12 bits, i.e., 111010111011. The authors studied the statistical analysis of the proposed work and they derive lower and upper bounds for test reduction.

Obviously, the order in which test vectors are scanned-in affects the amount of compression obtained using this scheme. For this reason, the authors try to find a suboptimal ordering of the test vectors by modelling the compression problem as a complete directed graph analogous to the travelling salesman problem. They solve the transformed problem by using two simple test ordering algorithms.

They present their results in terms of the reduction in the number of bits. Results are given for randomly generated test sets as well as deterministic test sets for some of the ISCAS85 benchmarks.

Variable-Length Reseeding

Hellebrand *et al.* [**?**] have presented a method to compress test cubes based on the reseeding of Multiple Polynomial LFSRs (MP-LFSRs). In their method, a group of concatenated test cubes with a total of s specified bits is encoded with approximately s bits LFSR specifying a seed and a polynomial identifier. Thus, the more unspecified bits a group has, the lesser bits needed for the LFSR. So, one of our proposed techniques can be used to relax the test set before the construction of an LFSR. The content of the MP-LFSR is loaded for each test group and has to be preserved during the decompression of each test cube within the group. Thus the implementation of the decompressor may involve adding extra flip-flops to avoid overwriting the content of the MP-LFSR during the decompression process.

Zacharia *et al.* [**?**] have avoided the overwriting problem by introducing an alternative to concatenation, i.e., *variable-length reseeding*. In their technique, deterministic patterns are generated by an LFSR loaded with seeds whose lengths may be smaller than the size of the LFSR. Allowing such shorter seeds yields higher compression rate even for test cubes

with varying number of specified bits. The decompression hardware is loaded for each test pattern. Hence, it is possible to implement the decompressor by using scan flip-flops as the state of the decompressor can be overwritten between applications of test cubes. As a result, the decompression can be implemented without any extra flip-flop.

Since most modern circuits feature multiple scan chains to reduce test application time, Zacharia *et al.* [**?**] have proposed compression schemes suitable for circuits with multiple scan chains.

Since seeds are of variable lengths, some extra information has to be maintained to specify the length of the seed. The test controller maintains the current length of the seed and one extra bit is padded to each seed to indicate when the current length should be increased. Since only one bit is used, the length is increased with a constant increment d. Using fixed increment requires some extra zeroes to be added to the seeds such that their lengths can always be expressed as $b + i \cdot d$, where d is the length of the shortest seed. However, the value of the increment can be chosen such that the number of extra zeroes is kept minimum.

In [**?**] and [**?**], three techniques are proposed for the implementation of the decompressor. The first one uses a two-dimensional hardware decompressor for multiple-scan chain designs. It exploits the existing scan flip-flops and the flip-flops of the PRPGs with the addition of few extra gates (XOR and AND gates). The goal of this implementation is to allow the decompression of test vectors with large number of specified bits while minimizing the area overhead.

The second implementation uses the embedded processor available in some core-designs to load and execute the decompression algorithm. In this way, no additional hardware is needed and hence no area overhead. Here, a program (or a microcode) is executed in the

embedded processor to read data from external memory to the local register file and decompress the data to the scan chains of the CUT. To reduce test application time, the number of instructions needed to decompress the data has to be minimized.

The third alternative is used with designs that include boundary scan chains. This is useful when CUTs are mounted in a board during testing.

Test Width Compression

Chakrabarty *et al.* [**?**] have proposed a technique to reduce the width of a pre-computed test set T_D (a matrix of size $m \times N$, where m is the number of vectors and N is the width of one vector). A Test Generation Circuit (TGC) is used to produce a compressed vector of width w that is less than the original test width N. Then, a decoder circuit is responsible for restoring the original vector. In other words, the deterministic test set T_D is generated first using conventional ATPG. Then, the TGC is used to compress it in order to reduce both timing and storage requirements. Different TGC implementations have been proposed, however, the most effective and widely used method is to combine LFSR with ROM.

This technique is based on the following definitions. Given a test set T_D (in the paper, T_D is referred to as a test matrix), two columns a and b of T_D are said to be *compatible* if for every row i, $a_i = b_i$ or one of them is a don't care x. The two columns a and b are said to be *inversely compatible* if for every row i, $a_i = \overline{b_i}$ or one of them is x. They are said to be *d-compatible* if there is no row in which both of them equal 1. A *maximal d-compatible* (MDC) class is the set of all columns in T_D that are pairwise d-compatible. In an MDC, there is at most one 1 in each row of the MDC. Therefore, it is possible to encode a row of an MDC by specifying the position of the 1. if no such 1 exists, then a row is encoded by 0.

So, for each row, the number of bits needed for encoding is $\lceil \log_2 (n+1) \rceil$, where n is the number of columns in the MDC.

To encode a test set T_D using this technique, a set M of k MDCs $\{C_1, C_2, \cdots, C_k\}$ is to be found such that:

1. Each column in T_D must appear in at least one MDC.
2. The width of all compressed vectors w is minimized, where $w = \sum_{i=1}^{k} \lceil \log_2 (n_i + 1) \rceil$.

The set M is called an optimal MDC cover. However, finding the best M is an NP-complete problem.

The following steps summarizes what this technique does to encode a test set T_D:

1. Reduce T_D by merging compatible and inversely compatible columns. This is done in hardware by assigning compatible columns to same output and inversely compatible columns to an inverter of the same output of the TGC.
2. Apply column complementation to reduce number of ones. If a column is complemented, the corresponding output of the TGC must be inverted.
3. Apply row complementation to reduce number of ones. Here, a redundant column is added to indicate whether a row is complemented or not. By XORing this column with the rows, the original set can be restored. However, careful computation must be done to compromise between compression and added overhead.
4. Compute a near-optimal MDC cover using some heuristic.

5. Encode rows of each MDC.

Simulation results on ISCAS89 benchmark circuits show that the technique achieves high compression ratios for partially specified test sets (40%-99%). However, if only fully-specified test set is provided, almost no compression is achieved. Thus, one of our proposed techniques may be applied to T_D in order to relax it, then the relaxed version can be used as an input to this technique to have a better quality compression.

Replacement Word

Jas and Touba [**?**] have proposed a compression/decompression scheme for embedded processors that is based on generating the next vector from the previous one by storing only the information about how the vectors differ. Each test vector is divided into fixed length blocks. The size of these blocks depends on the word size of the processor. Next vector is built from previous one by replacing the blocks in which they differ. Because of the structural relationship between the faults in a circuit, there will be a lot of similarity among the test vectors. Ordering can be applied to maximize compression ratio.

A *replacement word* is an encoded piece of information which tells the processor how to build the next vector from the previous one. Such word is composed of three fields. One field is called *last flag*, which is a single bit field. Another field, which is $\log_2\ N$ bits, is called the *block number*, where N is the number of blocks per test vector. A third field is a b bit field and is called *new block pattern*. The block number contains the address of the blocks to be replaced with the new pattern contained in the new block pattern field. If two test vectors differ in n blocks, then this information is encoded as a sequence of n

replacement words where the last field of the nth word is turned on (1). Other replacement words have their last field turned off (0). The block size b is chosen such that $1 + \lceil \log_2 N \rceil + b = W$, where W is the word size of the processor.

If a partially specified test set is provided, unspecified portions may be specified to minimize the number of replacement words. Experiments were performed with some ISCAS benchmarks. Compression ratios obtained range from 27% to 73%.

Statistical Coding

Jas *et al.* [**?**] introduced a statistical coding that is used to compress deterministic test data. The technique uses a modified version of Huffman coding to simplify the decoding process.

The idea can be summarized as follows. First, divide the test vectors into equal size blocks, each of size b. If the size of test vectors is not divisible by b, additional x's can be added to the beginning of the vectors (since shifting them to the scan chain will not affect the test as long as the final content is the same as the original vectors). Then, compute the frequency (number of occurrences) of each block. After that, divide the blocks into two sets: one which includes n most frequent blocks and the other includes the rest. Next, build a Huffman tree for the n blocks (the first set). The blocks are encoded as follows: if the block belongs to the first set, its codeword is obtained from the Huffman tree with a 1 preceding it. Otherwise, the block is not encoded. Instead, it is preceded by a 0 to indicate this situation.

Simplicity of the decoder together with good average compression ratio can be achieved by careful selection of the two important parameters b and n. The authors implement the decoder using a finite state machine of $n + b$ states.

Although Huffman coding gives better compression ratios, Jas *et al.* technique offers a simple-to-implement decoding circuit. The number of states in Huffman coding grows exponentially, while the number of states in Jas *et al.* technique grows linearly.

Results are provided for a subset of ISCAS85 and ISCAS89 benchmark circuits. The results indicate that this scheme can use a simple decoder to provide a compression ratio near that of an optimal Huffman code.

Run-Length Coding with Burrows-Wheeler Transformation

Ishida *et al.* [**?**] [**?**] used a modified version of run-length coding to encode columns of test data after performing a Burrows-Wheeler (BW) transformation on each column. Their technique is based on some observations on test data. For example, logic values of a small subset of primary inputs change for a block of test patterns, while other primary inputs are kept at constant logic values. Therefore, if the test set is viewed as a matrix, some columns change their values more frequently than others. This introduces us to a term called *activity*. The activity of a string of symbols S, $\alpha(S)$, is defined as the number of transitions on S. For instance, the string (aaabaaabcc) has an activity of 4. The authors also observed that active columns usually form cycles, where a cycle is a sequence of symbols that repeats more than once in a string. For example, the above string has a cycle (aaab) that repeats two times.

To exploit these two characteristics of test data, Burrows-Wheeler (BW) transformation and run-length coding are used. BW transformation is performed on a string S of length n as follows. First, form a matrix of size $n \times n$, the first row of the matrix is S, and the following rows are formed by left-rotating the previous row. Then the rows of the matrix are sorted lexicographically. To be able to recover S from the sorted matrix, we need to know the last

column L of the sorted matrix and the index I of the original string S in the sorted matrix. Now, L is the BW transform of S. However, the decoding operation is simpler and it does not involve sorting.

BW transformation usually results in a string that contain lesser number of runs, and thus better compression using run-length coding. However, this is not always the case.

The overall compression procedure proposed is as follows. The BW transformation is performed on a given test set, and a new hybrid data set is formed by collecting either the original column or the BW-transformed one, whichever has lower activity. Run-length coding is applied to columns with low activity, and GZIP compression is applied to columns with high activity.

The authors propose a more efficient run-length coding scheme. Consider the two consecutive runs (s, L_s) and (t, L_t), where s and t are the repeating symbols, and L_s and L_t are the run-lengths, respectively. The two runs make a transition from symbol s to symbol t. Suppose that s is known. Then the proposed encoding scheme is to compound the run-length L_s of the first run and the repeating symbol of the following run, t, into a single integer. Details are as follows. Let s be the repeating symbol, L be the length of its run, t be the next symbol, and M be the length of the whole string. The rule to compound the run-length L and the repeating symbol t is shown in Table **??**. For example, consider the encoding of $x1111xx$ whose length M is 7. The first run x is followed by another run 1111. The length of the first run $L = 1$, and the transition is from x to 1, so, using Table **??**, the compound integer is $L + M = 1 + 7 = 8$. The next run is 1111 which is followed by xx. The run length L is 4, and the transition is from 1 to x, thus the compound integer is $L = 4$. The last run xx doesn't make any transition, and hence no need to encode it.

Transition $s \rightarrow t$		Symbol t		
		0	1	x
Symbol s	0	-	L	$L+M$
	1	$L+M$	-	L
	x	L	$L+M$	-

Table 2.1: Encoding of run-length [**?**].

Figure 2.2: Next symbol of the BW run-length coding [**?**].

In what follows, a summary of the compression procedure is given. The test set is partitioned into equal size matrices D_i of size $M \times Q$, where M is the number of rows and Q is vector width. BW transformation is applied on each individual column and the activity of each column is computed before and after the transformation. Then, a new matrix E_i is built such that each column k of E_i is the BW transformation of the corresponding column in D_i if its activity is less than the original column and less than some threshold; otherwise, the original column is copied to E_i. The modified run-length coding is used to encode each column of E_i.

The decompression procedure is done according to the following steps:

1. Start with the symbol given in the encoded data.
2. The length of the current run = $i \bmod M$, where i is the corresponding integer given in the code for the run.
3. Let $j = \left\lceil \frac{i}{M} \right\rceil$, then the following symbol in the string is the jth symbol from the current symbol as shown in Figure **??**.

The authors compare their results with six other compression techniques, namely, Huffman coding [**?**], Arithmetic coding [**?**], UNIX utility "compress", GNU utility "gzip", LZSS [**?**],

and LZW [**?**]. Their technique performs the best among these techniques. The average compression ratio obtained was 31.5%.

Decompression Using Cyclical Scan Chains

Jas and Touba describe test vectors decompression technique in [**?**] via cyclical scan chains (CSCs). CSC decompression involves the use of two scan chains. One is the test scan chain (TSC) where the test vectors will be applied to the CUT, and the other is the CSC where the decompression will take place. The serial output of the CSC feeds the serial input of the TSC and also loops back and is XORed with the serial input of the CSC. Two requirements for the CSC: It must have the same number of scan elements as the TSC. And its contents must be preserved when applying a test vector to the CUT.

The CSC has the property that if it contains a test vector t_i, next vector t_{i+1} can be generated by feeding CSC with the difference vector $t_i \oplus t_{i+1}$. Careful ordering of the test vectors can maximize the number of zeroes in the difference vectors. For this reason, the authors applied the variable-to-block run-length coding compression algorithm, since data in which the probability of one value exceeds that of another can be efficiently compressed with run-length coding.

The authors tries to order the vectors in the test set to minimize the number of runs in the difference vectors. They solve this problem by forming a complete weighted graph and finding the minimum cost Hamiltonian path. Each node in the graph correspond to a test vector and is connected by a weighted edge to every other node in the graph. The weight on an edge between two nodes is computed by forming the difference vector and computing the number of bits needed to encode the difference vector. The minimum cost Hamiltonian

path corresponds to the optimal ordering of the test vectors to maximize the compression ratio.

Experiments were performed on ISCAS85 and large ISCAS89 benchmarks using two different codes: 2-bit code and 3-bit code. 3-bit code provide better compression ratios which range from 12% to 39%. The amount of compression could be much greater if test data is partially specified.

Compression Based On Golomb Coding

Chandra and Chakrabarty [**?**] have proposed a test compression scheme based on Golomb coding. Their proposed technique compresses the difference vectors T_{diff} rather than the original test vectors T_D. It assumes a decompression architecture similar to the CSC idea (refer to Section **??**).

After T_{diff} is generated from T_D, the next step is to select a Golomb code parameter m called the group size. Once m is determined (through experimentation), the runs of 0's in T_{diff} are mapped to groups of size m, each group corresponds to a run length. The number of such groups is determined from the length of the longest run in T_{diff}. The set of run lengths $\{0, 1, \cdots, m-1\}$ forms group A_1; The set $\{m, m+1, \cdots, 2m-1\}$ forms group A_2, ...etc. In general, the set $\{(k-1)m, (k-1)m+1, \cdots, km-1\}$ forms group A_k. Each group A_k is assigned a prefix of $(k-1)$ ones followed by a zero. This is denoted by $1^{(k-1)}0$. A group tail is also needed to differentiate between a group member. The tail ranges from 0 to $m-1$ and thus occupies $\lfloor \log_2 m \rfloor$ bits. The final codeword needed to encode a run of zeroes is composed of the group prefix and the group tail concatenated with each other. Figure **??** illustrates the construction of a Golomb code for $m = 2$. As an

Group	Run-length	Group prefix	Tail	Codeword
A_1	0	0	0	00
	1		1	01
A_2	2	10	0	100
	3		1	101
A_1	4	110	0	1100
	5		1	1101
A_1	6	1110	0	11100
	7		1	11101
...	...	...	...	...

Figure 2.3: An example of Golomb coding for $m = 2$.

example, consider T_{diff} = 001 00001 0001 1 0000001. The corresponding Golomb encode codewords are T_E = 100 1100 101 00 11100.

The authors proposes a simple and scalable decoder which is independent of both the core under test and the precomputed test set. Moreover, their decoder does not introduce a significant hardware overhead due to its small size. It can be efficiently implemented by a $\log_2 m$ bit counter and a finite state machine. The decoder decompresses the encoded test data and produces T_{diff}.

Experimental results on Golomb coding are given for ISCAS benchmarks with test sets generated using MinTest [**?**]. Results are also given for two industrial circuits with various test sequences. The compression ratios are calculated for different values of m. The maximum compression ratio was obtained for a group size of 512. Also, a comparison between Golomb coding and run-length coding for fully specified test sets is given.

Frequency-Directed Run-Length (FDR) Coding

Although pre-existing research on test data compression clearly demonstrated how compression offers a practical solution to the problem of reducing test data volume, the compression codes used were derived from other application areas. For example, statistical coding used in [**?**] is motivated by pattern repetition in large text files. Similarly, the run-length and Golomb codes used in [**?**, **?**] are more effective in encoding images. None of these codes where specifically tailored to exploit the special properties of the precomputed test data for logic circuits. While an attempt was made in [**?**] to customize the Golomb code by choosing an appropriate code parameter, the basic structure of the code was still independent of test set.

Chandra and Chakrabarty [**?**] have proposed a new class of variable-to-variable-length compression codes that are designed using the distributions of the runs of 0s in a typical test set. This way, the code can be tailored to SOC test data compression. The proposed codes are termed as Frequency-Directed Run-Length (FDR) codes.

An FDR code is a variable-to-variable-length which maps variable-length runs of 0s to codewords of variable length. It can be effectively used to compress both the test data itself T_D or the difference vectors T_{diff}. When compressing T_{diff} a CSC is assumed to exist. Compressing T_D requires replacing don't cares with 0s. This alternative, i.e., compressing T_D, is especially attractive since it eliminates the need for a CSC. The authors examined the distribution of the runs of 0s in typical test sets and they concluded the following:

- The frequency of runs of 0s of length l is large for $0 \leq l \leq 20$.
- The frequency of runs of 0s of length l is small for $l \geq 20$.

- Even in the range $0 \leq l \leq 20$, the frequency of runs of length l decreases rapidly as l increases.

An FDR code is constructed as follows. The runs of zeroes are divided into groups $A_1, \cdots, A_k$, where k is determined by the length l_{max} of the longest run ($2^k - 3 < l_{max} \leq 2^{k+1} - 3$). Note that a run of length l is mapped into group A_j, where $j = \lceil \log_2(l + 3) - 1 \rceil$. The size of the i^{th} group equals 2^i. This is unlike Golomb codes where the size of all groups is equal to m. Each codeword is composed of two parts: group prefix and tail - just like the case in Golomb coding. Figure **??** shows a construction of an FDR code. As clearly seen from the Figure, an FDR code has the following properties:

- For any codeword, the prefix and tail are of equal length.
- The length of the prefix of group A_i is i.
- For any codeword, the prefix is identical to the binary representation of the run-length corresponding to the first element of the group.
- The codeword size increases by two bits as we move from group A_i to A_{i+1}.

The authors present an analytical characterization of the amount of compression that can be expected when using FDR codes. Analytical results show that FDR codes are robust, i.e., they are insensitive to variations in the input stream. A decompression architecture is also presented for such codes. The decoder is simple and scalable, and independent of both the core under test and the precomputed test set. It is composed of a finite state machine, a k-bit counter, and a $\log_2 k$-bit counter which interact with each other in order to decompress the test data. Due to its small size, it does not introduce significant hardware overhead.

Group	Run-length	Group prefix	Tail	Codeword
A_1	0	0	0	00
	1		1	01
A_2	2	10	00	1000
	3		01	1001
	4		10	1010
	5		11	1011
A_3	6	110	000	110000
	7		001	110001
	8		010	110010
	9		011	110011
	10		100	110100
	11		101	110101
	12		110	110110
	13		111	110111
...	...	...	...	...

Figure 2.4: An example FDR coding [**?**].

Experiments were performed on ISCAS 89 benchmark circuits. The results show that FDR codes outperform regular Golomb codes for test data compression.

Extended FDR Coding

Elmaleh and Alabaji [**?**] have analyzed the test data and found that test sets contain a large number of runs of 1's in addition to runs of 0's. By considering both types of runs, the total number of runs will decrease, which could result in higher test data compression. They have supported this observation by experimental analysis of test data for the largest ISCAS 85 and full-scanned versions of ISCAS 89 circuits. They have generated their test sets using MinTest [**?**], using both static and dynamic compaction. Test sets generated based on static compaction were relaxed, as this has the advantage of keeping unnecessary assignments

as x's, which enables higher compression. Given a relaxed test set, techniques based on encoding only runs of 0's fill all the x's with 0's to reduce the number of runs that need to be encoded. However, to encode both runs of 0's and 1's in a test set, x's are filled with 1's if they are bounded by 1's from both sides, otherwise they are filled by 0's. This results in a reduction in the total number of runs that need to be encoded.

The authors extended the FDR coding to encode both runs of 0's and 1's based on adding an extra bit to the beginning of a code word to indicate the type of run. If the bit is 0, this indicates that the code word is encoding a run of type 0, otherwise it encodes a run of type 1. It should be observed that this code, called Extended FDR (EFDR), is a direct extension to the FDR code shown in Figure **??**. However, in the EFDR, runs of length 0 do not exist because both runs of 0's and runs of 1's are encoded. Note that runs of 0's are strings of 0's followed by a 1, while runs of 1's are strings of 1's followed by a 0, i.e., runs of 1's of length i are the complement of runs of 0's of the same length, and vice versa. As with FDR code, in this code when moving from group A_i to group A_{i+1}, the length of code words increases by two bits, one for the prefix and one for the tail. Runs of length i are mapped to group A_j, where $j = \lceil \log_2(i+2) \rceil - 1$. The size of the i^{th} group is $2i + 1$.

Based on experimental results on ISCAS benchmark circuits, the EFDR code outperforms the FDR code and results in significant increase in test data compression ratio for several circuits, improving the compression ratio from 19.36% to 80.31% for one of the benchmark circuits.

Compression Based On Geometric Shapes

Elmaleh *et al.* [**?**] have proposed a novel and efficient test compression technique, which is based on encoding the 0's and the 1's of the test set into the four primitive geometric shapes: points, lines, triangles, and rectangles. These basic shapes are the most frequently occurring shapes in test sets.

The compression algorithm is composed of three phases. First, the test vectors are sorted. This step has an important impact on the compression ratio. The purpose is to generate clusters of 0's or 1's in such a way that it may partially or totally be fitted in one or more geometric shapes. In their work, they have applied a simple correlation based sorting algorithm.

Phase two takes care of sorted test data partitioning. A set of sorted test vectors is represented as a matrix, M, of size $R \times C$, where R is the number of test vectors and C is the width of a test vector. The test data is partitioned into $L \times K$ blocks each one is $N \times N$ bits, such that $L = \lceil \frac{R}{N} \rceil$ and $K = \lceil \frac{C}{N} \rceil$.

Finally comes the encoding phase, which is applied to each one of the $N \times N$ blocks generated in the previous phase. In this phase, the shapes are extracted first. Then a covering problem is solved to minimize the number of shapes, and thus the number of bits, needed to encode the test data. This is done once for the 0's and once for the 1's, and the encoder will choose whichever produces better result.

The authors also proposes a simple and straightforward decoder that is assumed to be executed in an embedded processor on a chip. Simulation results show that the decoding algorithm is very fast and that the decoding time for the test sets experimented with was in

fractions of a second for each test set.

Experiments were performed on a number of ISCAS85 and full-scan versions of ISCAS89. Based on the results, this method has an average compression ratio of 76%, based on highly compacted test sets.

2.2.3 Test Power Reduction

Excessive switching activity during scan testing can cause average power dissipation and peak power during test to be much higher than during normal operation. This can cause problems both with heat dissipation and with current spikes. Compacting scan vectors greatly increases the power dissipation for the vectors (generally the power becomes several times greater). The compacted scan vectors often can exceed the power constraints and hence cannot be used. Sankaralingam *et al.* observed in [**?**] that by carefully selecting the order in which pairs of test cubes are merged during static compaction, both average power and peak power for the final test set can be greatly reduced. They presented a static compaction procedure that can be used to find a minimal set of scan vectors that satisfies constraints on both average power and peak power.

During static compaction, test cubes are merged and unspecified x values get specified. Let $Tran\_Count(A)$ be the number of transitions in test cube A, and $Tran\_Count(B)$ be the number of transitions in test cube B, then if test cube A and B are merged to form test cube C, then the number of transitions in test cube C, $Tran\_Count(C)$, will be greater than or equal to the maximum of $Tran\_Count(A)$ and $Tran\_Count(B)$. In some cases, the number of transitions in the merged test cube can be much larger than in either of the original test cubes. Consider the case where the test cube 0X0X0X is merged with X1X1X1

to form 010101. The original test cubes had 0 transitions, but the merged test cube has 5 transitions. Conventional static compaction procedures randomly merge compatible test cubes. However, merging some pairs of test cubes can result in a test cube with a large number of transitions whereas merging others would not increase the number of transitions by much. If the wrong test cubes are merged, the power can increase dramatically. So the idea behind the authors' proposed static compaction procedure is to direct the process of selecting which test cubes to merge in a way that avoids generating merged test cubes with large numbers of transitions, while trying to minimize power in the final test set. Their proposed procedure for static compaction is as follows:

Create the cost graph
while there are vectors which can be combined **do**
Select a pair to be combined
Combine the selected pair
Update the cost graph
end while

The procedure begins by forming a cost graph. This graph is constructed as follows. There is one node for each test cube. For each pair of compatible test cubes (a, b), an edge is placed between the corresponding nodes. The weight attached to the edge is the increase in power that results from removing a and b from the test set and applying the merged vector $a \cap b$ instead. In each iteration of the procedure, a pair of test cubes to be combined is selected. The objective is to pick pairs of test cubes so that the average power or peak power is minimized. The problem is complex as the graph changes each time a test cube pair is merged because two nodes are replaced by a single node. The procedure chooses the test cube pair using a greedy heuristic. The pair of nodes with the smallest edge weight

is selected in each iteration. When two test cubes are combined, the graph can be quickly updated. The old nodes and their edges to other nodes are no longer valid. The two nodes are removed and a new node representing the combined vector is created. The procedure continues until no two vectors can be combined. If there is a constraint on average power during test for some CUT, the average power can be monitored during each iteration of the static compaction procedure and the procedure can stop if the average power reaches a certain threshold.

Experiments on some of the largest ISCAS 89 circuits are provided. For each of the circuits, two plots are generated. The first is a plot of average power versus the number of test vectors. As the number of vectors decreases the average power increases. This is because the average number of transitions per vector increases due to the merging of vectors. For all circuits, the average power of the authors' proposed static compaction procedure is much less than that of conventional (random) static compaction. All the circuits have a characteristic "knee" in the graph where the average power rises rapidly with a small decrease in the number of test vectors. Given plots like these, the designer can decide to sacrifice a small amount of compression and gain a lot in power savings. The other plot is the peak power of the test set versus number of vectors. The peak power is the maximum number of transitions that occur in a single test vector in the entire test set at that stage. The average number of transitions per vector increases when vectors are combined. Hence as expected, the peak power increases as the number of vectors decrease. For all circuits the authors' proposed static compaction procedure has much lower peak power compared to conventional static compaction.

2.3 Existing Solutions of the Test Relaxation Problem

2.3.1 Bitwise Relaxation (BR) Method

We begin with the BR method. Although it is not considered as a practical solution to the problem, it is described here because we have compared it with our proposed techniques. In the early stages of this work, there was not any practical solution to the relaxation problem. However, a recent work presented by Kajihara and Miyase [**?**] that is parallel to our research, has emerged. As mentioned earlier, no comparison is provided with this work due to the unavailability of their experimental test sets. In subsequent discussions, this technique is referred to as the KMR (Kajihara-Miyase Relaxation) technique.

Algorithm **??** outlines the BR algorithm. Notice that the purpose of this algorithm is to obtain a solution, not necessarily an optimal solution. The Algorithm takes a fully specified test set (denoted by T) and tries to update it into a relaxed version.

Algorithm 2.5 Bitwise relaxation method.

```
for every bit b in T do
    b ← x
    Fault simulate T.
    if the fault coverage is reduced then
        Restore the original value of b
    end if
end for
```

The algorithm examines every bit of the test set in turn. If the examined bit contributes to the fault coverage, it is kept unchanged. Otherwise, it is turned into an x. The fault coverage is calculated using a fault simulator. In our implementation, HOPE [**?**] is used for fault simulation purposes.

The quality of the relaxation obtained using this technique depends on the following factors:

1. The ordering of test vectors in T.

2. The order in which primary inputs are traversed.

Obviously, this Algorithm has a complexity of $O(nm)$ fault simulation runs, where n is the number of test vectors, and m is the number of bits in each test vector. In each fault simulation run, only the detected faults by a particular vector are simulated. Obviously, this technique is impractical for large circuits.

2.3.2 The KMR Technique

Now, a description of the KMR technique is given. The input test set is denoted by T and the output (relaxed) test set is denoted by T'. The basic idea is that essential faults of $t_i \in T$, have to be detected by $t'_i \in T'$. However, nonessential faults of t_i do not have to be detected by t'_i because they have a chance to be detected by another vector.

Algorithm **??** outlines the main steps of the KMR technique. For every test t_i, the algorithm proceeds as follows. Essential faults of t_i are collected. Necessary input values to detect essential faults are marked. This is done by ATPG implication and justification procedures with minor modifications. The modification makes sure that the implied or justified values don't conflict with the implied values of the applied test pattern t_i. Thus, no backtracking exists. That's why the authors refer to these procedures as *limited implication* and *limited justification* procedures. After necessary inputs are marked, unmarked inputs are turned

Algorithm 2.6 An outline of the KMR technique [**?**].

for every test vector $t_i \in T$ **do**
  Logic simulate t_i
  $F =$ collect_essential_faults(t_i)
  $t'_i =$ find_value(F)
  fault_simulate(t'_i)
end for

for every test vector $t_i \in T$ **do**
  Logic simulate t_i
  $G =$ collect_undetected_faults(t_i)
  $t'_i +=$ find_value(G)
  fault_simulate(t'_i)
end for

for every test vector $t_i \in T$ **do**
  Logic simulate t_i
  $H =$ collect_undetected_faults(t_i)
  $t'_i =$ extended_find_value(H)
  fault_simulate(t'_i)
end for

Return T' composed of t'_i

into don't cares (x's). The resulting test vector is copied into t'_i. Fault simulation of t'_i is performed to drop all the detected faults.

This is followed by another loop to ensure the detection of nonessential faults. For every test vector t_i, undetected faults of t_i are first collected. Then, try to imply and justify the values needed to detect these faults under t'_i. Then, t'_i is updated correspondingly. For some reason, some of the faults will still be undetected, so fault simulation is performed for another time to drop detected faults.

The final loop deals with the faults that are still undetected after the second loop is over, even though these faults are treated explicitly. The reason why such faults occur, is that the used implication and justification look only at the fault-free values, they don't look at the faulty values. While the fault-free value is implied and justified correctly, the faulty value might get masked along the propagation path producing an unspecified faulty value (x). For this reason, the author proposes *extended justification* procedure, which can be described as follows. Whenever an output is to be justified, simply justify all its inputs. For this reason, unnecessary assignments might result from extended justification. The authors recommend that extended justification should not be performed for many faults, otherwise the quality of the relaxation might become very poor. So, after the second loop is over, traverse all t_i looking for undetected faults due to masking. Perform extended justification and update t'_i accordingly. Now, T' which is composed of t'_i, contains the relaxed test vectors and maintains the same fault coverage as T.

Worst-Case Analysis Of The KMR Algorithm

An analysis of the worst-case behavior of the KMR Algorithm is given here. The following assumptions are made:

- Gate processing is considered to be the basic operation. Moreover, gate processing can be any one of the following operations: forward/backward implication, and justification. Thus, the total number of basic operations is simply the sum of the number of forward/backward implications, and the number of justifications.
- Fault simulation is assumed to be serial, i.e., one fault is simulated at a time.
- To simplify the analysis, fault simulation without fault dropping is assumed, i.e., when a fault is detected by a test vector, it will still be simulated under later vectors.
- Let N_T be the number of test vectors in the input test set.
- Let N_F be the number of faults in the fault list.
- Let N_G be the number of gates in the input circuit.

Given these assumptions, an upper-bound on the number of basic operations in the first loop can be calculated as follows:

- Logic simulation requires at most N_G basic operations.
- Collecting essential faults of a test vector requires at most $N_F N_G$ basic operations.
- Performing implication and justification requires at most N_G basic operations.

- Fault simulating a test vector requires at most $N_F N_G$ basic operations.
- The whole thing is done N_T times.

In conclusion, an upper-bound on the number of basic operations in the first loop is

$$(2 + 2N_F)\, N_G N_T$$

In a similar way, the number of basic operations in the second loop is at most

$$(2 + N_F)\, N_G N_T$$

Similarly, the number of basic operations in the third loop is at most

$$(2 + N_F)\, N_G N_T$$

Thus, an upper-bound on the number of basic operations in the KMR Algorithm is

$$2\,(3 + 2N_F)\, N_G N_T \tag{2.1}$$

Thus, the KMR Algorithm is $O(N_G N_F N_T)$ basic operations. However, in the comparison with our proposed techniques, we will refer to Equation **??** rather than the Big-O notation, since it is more accurate.

2.4 Concluding Remarks

In this chapter, we reviewed some basic background information including the critical path tracing Algorithm. After that, we listed some techniques which might benefit from test relaxation, i.e., test compaction, test compression, and test power reduction. Also the BR and the KMR techniques have been reviewed in this Chapter.

Chapter 3

The Proposed Relaxation Techniques

In this Chapter, we present our proposed test relaxation techniques. As mentioned earlier, the first proposed technique is based on the CRIPT Algorithm. It is referred to as the CRIPTR technique. As with CRIPT, this technique is not exact in the sense that the fault coverage might be reduced after relaxation. However, as will be shown from experimental results (in the next Chapter), the drop in the fault coverage is small for most of the circuits.

In the second proposed technique, the relaxed test set maintains the same fault coverage of the original test set. This technique simply identifies all the newly detected faults under a given test vector, by the help of a fault simulator, and marks all the lines whose values are required to detect the faults. The unmarked inputs are relaxed. As mentioned before, this technique is referred to as the SVR technique.

The third technique is an improvement over the SVR technique. This is referred to as the Two-Values Relaxation (TVR) Algorithm. This name emerged because, in the TVR

Figure 3.1: An example circuit.

Algorithm, one keeps track of both the fault-free and the faulty values. This is unlike the CRIPTR and the SVR techniques.

Throughout this Chapter, the following conventions are used. To indicate that a line l is stuck at value v, we use the notation l/v. The notation $A = x/y$ is used to indicate that the fault-free value of line A is x, and the faulty value of line A is y. For example, $A = 0/1$ means that the fault-free value of line A is 0, and the faulty value of A is 1. When it is said that line l is required, this means that the value on line l is required.

The rest of this chapter is organized as follows. Section 3.1 discusses the CRIPTR technique. The SVR technique is discussed in Section 3.2. Then, the TVR technique is presented in Section 3.3. In Section 3.4, selection criteria are discussed. A theoretical comparison between our proposed techniques and the KMR technique is given in Section 3.5.

3.1 The CRIPTR Technique

3.1.1 An Illustrative Example

We now demonstrate our first proposed test relaxation technique by an example.

Example 3.1:

Consider the circuit shown in Figure **??**. Suppose that we apply the test vector 00000. Under this test, lines $G6$, $G5$, $G1$, $G4$, $B2$, and B are critical. So, the faults $G6/0$, $G5/0$, $G1/1$,

$G4/0$, $B2/1$, and $B/1$ are detected under this test. Assume that the newly detected fault is only $B/1$. For this fault to be detected, it has to be activated (excited) and propagated to the primary output $G6$. The assignment $B = 0$ excites the fault. The assignments $G3 = 0$ and $G1 = 0$ are required for fault propagation. The assignment $B = 0$ is already satisfied because B is a primary input. The assignment $G3 = 0$ can be satisfied by either one of the two assignments $C = 0$, or ($D = 0$ and $E = 0$). If we choose to satisfy $G3 = 0$ by the assignment $C = 0$, then the assignments $D = 0$ and $E = 0$ are no longer necessary, and this implies that we can relax CDE to $0xx$. Similarly, if we choose to satisfy $G3 = 0$ by the assignments $D = 0$ and $E = 0$, then $CDE = x00$. So, there might exist more than one relaxed version of a given fully specified test vector, and some versions might have more unspecified bits than others.

The other requirement for fault propagation, which is $G1 = 0$, appears to be already satisfied because we already have marked the assignment $B = 0$ as required, and this assignment produces $G1 = 0$. This results in relaxing the input A since it is no longer necessary. But this is incorrect. To show that this relaxation is not correct, assume that stem B is faulty, i.e., $B = 0/1$ (i.e., the fault-free value is 0 and the faulty value is 1). In this case, if line A is relaxed, the fault on the stem will not propagate to the output. It will be masked by the x value on line A, producing the value $1/x$ on the output $G6$. The problem occurs because we justified the requirement on line $G1$ from line $B1$, which is reachable from the critical stem B. Justifying a required value from a reachable line, guarantees that the required value is satisfied in the fault-free machine but not in the faulty machine. This problem can be avoided by justifying the required value from an unreachable line. This guarantees that the value will be satisfied for both the fault-free and the faulty machine. For this example, the required value on line $G1$ has to be satisfied by marking line A as required, resulting

in the test vector $ABCDE = 100xx$, or $ABCDE = 10x00$. This example shows that we need to identify reachable lines before justifying the requirement list.

After this introductory example, the CRIPTR Algorithm is described next.

3.1.2 The CRIPTR Algorithm

Algorithm 3.1 Main part of the CRIPTR Algorithm.

```
1:  for every test vector t do
2:    for every output o do
3:      Extend(o)
4:      while StemsToCheck is not empty do
5:        s ← highest level stem in StemsToCheck
6:        Remove s from StemsToCheck
7:        if Critical(s) then
8:          if the fault f on s is newly detected then
9:            add f to NDF
10:           add s to CS
11:         else
12:           add s to CandidateStems
13:         end if
14:         Extend(s)
15:       end if
16:     end while
17:     AddCandidateStems()
18:     MarkReachableLines()
19:     MarkRequiredLines()
20:     Mark All the lines as non-critical & unreachable
21:   end for
22:   Output relaxed vector
23:   Mark all Lines as non-required
24: end for
```

Algorithm **??** shows a general outline of the proposed test relaxation technique. Initially, all the lines are marked as non-critical, unreachable, non-required. For every primary output

o under the test vector t, the Algorithm performs critical path tracing while storing the newly detected faults in the NDF list, the critical stems whose faults are newly detected in the CS list, and the critical stems whose faults are previously detected (through a previous output or vector) in the $CandidateStems$ list. We have chosen the name $CandidateStems$ because any stem in the $CandidateStems$ list is a candidate to be added to the list CS if it satisfies one condition: there is at least one newly detected fault passing through it. The procedure $AddCandidateStems$, shown in Algorithm **??**, checks, for every stem s in the $CandidateStems$ list, whether s satisfies the condition or not. If s satisfies the condition, it is inserted in the CS list. Otherwise, it is ignored. One can observe that the CS list consists of two kinds of critical stems: the first kind is a critical stem which has a newly detected fault on it, and the other kind is a critical stem whose fault was previously detected but there is a newly detected fault (coming from another line) that passes through it. Both kinds are needed in the reachability analysis.

The $Extend$ procedure is the same as the one given in [**?**], but it does one extra job, namely adding newly detected faults to the NDF list. The $Critical$ function is exactly the same as the one given Algorithm **??**. This function forms the bottleneck of our implementation, since it often does a fault simulation when it is called.

Once the CS and NDF lists are constructed, the Algorithm marks reachable lines by calling the procedure $MarkReachableLines$ shown in Algorithm **??**. Then the Algorithm justifies the requirements by the procedure $MarkRequiredLines$. The last statement in the inner loop is a re-initialization of the criticality status and reachability status of the lines. After the inner loop is finished, the relaxed vector is ready and is printed out. For the next vector, we re-initialize the requirement status of all the lines.

Algorithm 3.2 AddCandidateStems()

```
1: while CandidateStems is not empty do
2:     let s be an element of CandidateStems
3:     delete s from CandidateStems
4:     if a newly detected fault passes through s then
5:         add s to CS
6:     end if
7: end while
```

Reachability Analysis

This phase takes the list CS as an input. The purpose of this phase is to mark the lines that are reachable from at least one element of the list CS as reachable.

Algorithm 3.3 MarkReachableLines() - CRIPTR Algorithm version.

```
1: initialize the event list E
2: for every element s in CS do
3:     mark fanouts of s as reachable from s
4:     add fanouts of s to E
5:     while E is not empty do
6:         l ← element in E with minimal level
7:         remove l from E
8:         if Reachable(l, s) then
9:             if l is not a fanout stem then
10:                Mark l as reachable from s
11:                Add output of l to E
12:            else
13:                for every fanout branch b of l do
14:                    Mark b as reachable from s
15:                    Add output of b to E
16:                end for
17:            end if
18:        end if
19:    end while
20: end for
```

Algorithm **??** is an event driven Algorithm for marking reachable lines. The function

$Reachable(l, s)$ in the Algorithm returns true only if the fault effect in stem s reaches the line l. The following lemmas provide the rules used by the function $Reachable(l, s)$.

Lemma 3.1: Let l be an output of an AND, NAND, OR, or NOR gate. Then, if there is at least one of the inputs of l that has a don't care value, then l is unreachable from any stem except itself, if it is a stem.

Proof: Let us suppose that there are some inputs of l that are reachable from a stem s. Let us suppose further that the fault effect on these reachable inputs propagate to l in the absence of an x input. In this case, if one of the inputs of l is x, then this x will either mask the fault-free or the faulty value of l depending on which one is the non-controlling value. Assuming that c is the output-controlling value of l, then this x input produces either c/x or x/c, and in both cases the fault effect of stem s is not propagated to l. Thus, l is unreachable from any stem except the trivial case when it is reachable from itself, assuming that it is a stem. Q.E.D.

Lemma 3.2: Let l be an output of an AND, NAND, OR, or NOR gate. Suppose further that all inputs of l have specified values, i.e., none of the inputs of l has an x value. Then l is reachable from stem s iff exactly one of the following conditions is satisfied:

1. All controlling value inputs of l are reachable from stem s.
2. At least one of the inputs of l is reachable from stem s given that all the inputs of l have non-controlling values.

Proof: This lemma contains four statements to be proved. Let's prove them one by one.

1. **l has controlling value inputs and is reachable from stem $s \Rightarrow$ All controlling value inputs of l are reachable from stem s.**

   It is given that l is reachable from stem s. This implies that the fault on stem s (call it f) propagates to l implying that f has propagated through the inputs of l. It is given that l has a controlling value, and this implies that at least one of the inputs of l has a controlling value. Let's suppose that l has some non-controlling value inputs and that f reaches only (some or all of the) non-controlling value inputs of l. This implies that f will be masked by the controlling value inputs of l and that l is not reachable from s. This assumption contradicts with the given information, and thus is incorrect. Another possible assumption is that f reaches at least one controlling value input and at least one non-controlling value input of l. This also implies that f will be masked and that l is unreachable. Thus, this is an incorrect assumption. A third possibility is that f reaches only part of the controlling value inputs. In this case, f will be masked as well by the unreachable controlling value inputs and l will be unreachable implying that this assumption is incorrect. Thus, the only remaining assumption is that f reaches all controlling value inputs of l. In this case, f will propagate to l and no masking will occur. Thus, if there are inputs of l with controlling value and l is reachable from stem s, then all controlling value inputs of l are reachable from stem s.

2. **All controlling value inputs of l are reachable from stem $s \Rightarrow l$ is reachable from stem s.**

   This is obvious, because if f reaches all controlling value inputs of l, then f will propagate to l. Thus, by definition, l is reachable from stem s.

3. **All inputs of l have non-controlling values and l is reachable from stem $s \Rightarrow$ At least one of the inputs of l is reachable from stem s.**

   It is given that all inputs of l have non-controlling values. Since f reaches l, it has to propagate through its inputs. It is sufficient for f to propagate through one input of l because fault masking can never occur in the absence of controlling value inputs. Thus, at least one of inputs of l has to be reachable from stem s.

4. **All inputs of l have non-controlling values and at least one of the inputs of l is reachable from stem $s \Rightarrow l$ is reachable from stem s.**

   It is given that all inputs of l have non-controlling values. It is also given that f reaches at least one input of l. This implies that f will propagate to l and thus, by definition, l is reachable from stem s. Q.E.D.

Lemma 3.3: Let l be an output of a 2-input XOR/XNOR gate. Then l is reachable from stem s iff only one input is reachable from stem s, and the other input does not have an x value.

Proof: If l is reachable from stem s, then the fault on stem s, say f, reaches l. Thus, either one or both inputs of l are reachable from stem s. Suppose that f reaches both inputs of l. If both inputs of l have the same parity, then the fault-free and the faulty values of l will be 0 for the XOR and 1 for the XNOR, and this contradicts with l being reachable from stem s. On the other hand, if both inputs have different parities, then the fault-free and the faulty values of l will be 1 for the XOR and 0 for the XNOR, and this also contradicts with l being reachable from stem s. Thus, only one input of l should be reachable from stem s and the other input must not be an x to avoid fault masking.

Let's prove the other part of this lemma. If only one input of l is reachable from stem s and the other input is not an x, then the fault effect of f propagates to one input of l and the other input is either a logic 0 or a logic 1. In both cases, the fault effect of f will propagate to l implying that l is reachable from stem s. Q.E.D.

Algorithm 3.4 MarkRequiredLines()

```
1: Initialize the requirement list L
2: for every fault f in NDF do
3:     Let f be the fault on line l
4:     Add l to L
5:     ForwardTrace(l)
6: end for
7: for every line l in L do
8:     justify(l)
9: end for
```

Requirement Analysis

Algorithm **??** is a general outline of the requirement analysis phase. Initially, all the lines in the circuit are marked as non-required. After that, we perform a forward tracing step for every element in the list NDF. The purpose of this step is to identify paths through which the faults belonging to NDF propagate to an output. This is done by tracing the critical path from the line that has the newly detected fault until we reach a primary output, adding the side inputs of every sensitive input in that path to the requirement list, and marking the lines along that path and its side inputs as required. This step is outlined in Algorithm **??**. After this step is over, we will have a requirement list L to be justified.

Algorithm **??** is the value justification Algorithm used. Assume that line l is to be justified. If l is a PI, the Algorithm marks it as required and returns. If l is a single-input, XOR or

Algorithm 3.5 ForwardTrace(l)

```
1: if l is not an output of the circuit then
2:    if l is a stem then
3:       for every critical fanout branch b of l do
4:          Add side inputs of b to L
5:          Let j be the output of b
6:          ForwardTrace(j)
7:       end for
8:    else
9:       Add side inputs of l to L
10:      Let j be the output of l
11:      ForwardTrace(j)
12:   end if
13: end if
```

XNOR gate, all the values on l's inputs have to be justified. Similarly, all the values on the inputs of l have to be justified if l has a non-controlling value (assuming 0-inversion). However, if l has a controlling value, then we need to check if it has an unreachable input with a controlling value. If it has, then it is sufficient to justify the value using that unreachable input. Otherwise, we check whether l is reachable or not. If it is not reachable, then it is sufficient to justify only the reachable lines to preserve the fault-free and the faulty values of l. In this case, both values of l will be the same since l is unreachable. Otherwise, all the values on the inputs will be justified. The last two situations appear when l can only be justified from a reachable line. Note that in justifying a required controlling value, there could be several unreachable inputs with controlling value. In this case, priority is given to an input that is already marked as required. Otherwise, cost functions are used to guide the selection.

Algorithm 3.6 justify(l) - CRIPTR Algorithm version.

```
1: if if l is a PI then
2:     mark l as required
3: else if l is an output of a single-input, XOR, XNOR gate, or l has a non-controlling
   value then
4:     for every input j of l do
5:         justify(j)
6:     end for
7: else if there is an unreachable input line j of l with controlling value then
8:     justify(j)
9: else if l is unreachable then
10:    for every reachable input j do
11:        justify(j)
12:    end for
13: else
14:    for every input j of l do
15:        justify(j)
16:    end for
17: end if
```

3.1.3 Worst-Case Analysis Of The CRIPTR Algorithm

The same assumptions made for the worst-case analysis of the KMR technique are assumed here as well. Additional assumptions are as follows:

- Suppose that N_S is the total number of stems in the input circuit.
- Suppose also that N_O is the number of primary outputs in the input circuit.

The main part of the CRIPTR Algorithm performs the following subtasks $N_O N_T$ times:

1. **Logic simulation of the test vector under consideration:** This requires at most N_G basic operations.

2. **Critical path tracing:** This includes fault simulation of stems to check for criticality. Thus, the number of gates to be processed in this step can not exceed $N_G + N_S N_G$.

3. **Adding candidate stems:** The number of candidate stems can not exceed N_S. Thus, the number of gates to be processed here can not exceed $\frac{N_F}{N_T} N_S N_G$, where $\frac{N_F}{N_T}$ accounts for the average number of faults detected per test vector.

4. **Marking of reachable lines:** An upper-bound on the number of gates to be processed here is $N_S N_G$.

5. **Marking of required lines:** An upper-bound on the number of gates to be processed here is $\frac{N_F}{N_T} N_G$.

Summing up these terms results in the following upper-bound on the number of basic operations in the CRIPTR Algorithm

$$\left(1 + \left(2 + \frac{N_F}{N_T}\right) N_S + \frac{N_F}{N_T}\right) N_G N_O N_T \tag{3.1}$$

3.2 The SVR Technique

The main advantage of the SVR technique over the CRIPTR technique is that the SVR technique is based on an exact fault simulation, i.e., the fault coverage of the input test set is maintained in the output relaxed test set, unlike the CRIPTR technique. Algorithm **??** outlines the main part of the SVR Algorithm. For every test vector t, fault simulation is carried out to identify the newly detected faults. Then, for every newly detected fault f, the

Algorithm 3.7 Main part of the SVR Algorithm.

```
1: for every test vector t do
2:    Fault simulate the circuit under the test t
3:    for every newly detected fault f do
4:       BuildRequirementList(f)
5:       for every line l in L do
6:          justify(l)
7:       end for
8:       Mark all lines as unreachable
9:    end for
10:   Output relaxed vector
11:   Mark all Lines as non-required
12: end for
```

requirement list L of f is built, and justified. In what follows, each one of these tasks is explained.

Algorithm **??** builds the requirement list L for a given fault f. Assuming l is the faulty line, the Algorithm works as follows. First, line l is added to the requirement list (to ensure fault activation). Then, a while loop is entered. After the loop is over, line l is either an output or a fanout stem. In either case, the requirements for propagating fault f to line l are stored in L. However, if l is an output, the Algorithm terminates, because the requirements to detect f are completed. This situation occurs when the propagation path of f does not contain any fanout stem. Otherwise, an event-driven procedure is called for marking reachable lines from the fanout stem l, i.e., $MarkReachableLines$ shown in Algorithm **??**. This Algorithm is similar to Algorithm **??** with two main differences. One difference is that Algorithm **??** marks reachable lines for all the faults in the list CS, whereas Algorithm **??** performs reachability with respect to one line only. Another difference is that Algorithm **??** returns as soon as it finds an output with the output being the returned value. However, this is not the case in Algorithm **??** where all reachable lines of all the

elements in the list CS are marked. Algorithm **??** returns an output o in which f propagates to. The $BuildRequirementList$ procedure continues by tracing the propagation path of f starting from the output o. It adds all unreachable inputs in the propagation path of f to the requirement list L (to ensure fault propagation).

Algorithm 3.8 BuildRequirementList(f)

```
1: Let l be the line that has the fault f
2: Add l to L
3: while l is not a fanout stem nor a primary output do
4:    Add side inputs of l to L
5:    l ← output of l
6: end while
7: if l is a primary output then
8:    Return
9: end if
10: Let s = l
11: o ← MarkReachableLines(l)
12: Let E ← ∅
13: Add o to E
14: while E ≠ ∅ do
15:    Let j be an element in E
16:    Remove j from E
17:    for every input i of j do
18:       if i ≠ s then
19:          if i is reachable then
20:             Add i to E
21:          else
22:             Add i to L
23:          end if
24:       end if
25:    end for
26: end while
```

In the $MarkReachableLines$ Algorithm, once an output is reached, the loop is over and that output will be returned. The function $Reachable(l, j)$ is exactly the same as the one presented in the CRIPTR Algorithm . For value justification, Algorithm **??** is used.

Algorithm 3.9 MarkReachableLines(l) - SVR Algorithm version.

```
1: Let E ← ∅
2: Let j ← l
3: Add l to E
4: while E ≠ ∅ do
5:    Let l ← element in E with minimal level
6:    remove l from E
7:    if Reachable(l, j) then
8:       if l is a primary output then
9:          Mark l as reachable from j
10:         Return l
11:      else if l is not a fanout stem then
12:         Mark l as reachable from j
13:         Add output of l to E
14:      else
15:         for every fanout branch b of l do
16:            Mark b as reachable from j
17:            Add output of b to E
18:         end for
19:      end if
20:   end if
21: end while
```

3.2.1 Worst-Case Analysis Of The SVR Algorithm

The same assumptions made for the worst-case analysis of the CRIPTR and the KMR techniques are assumed here as well.

For every newly detected fault f by a test vector t, the SVR technique builds the requirement list of f, and justifies these requirements. Identifying newly detected faults requires at most $N_F N_G$. Building the requirement list and justifying these requirements for one fault requires at most N_G basic operations, and for one test vector, at most N_F faults are detected. Thus, an upper-bound on the total number of basic operations is

Figure 3.2: A limitation of the CRIPTR and the SVR Algorithms.

$$2N_G N_F N_T \tag{3.2}$$

3.3 The TVR Technique

The following example shows a limitation of the CRIPTR and the SVR Algorithms and illustrates how to overcome this limitation by utilizing the TVR Algorithm.

Example 3.2:

We illustrate here that it is possible in some cases to justify a value from a reachable line without any drop in fault coverage. Consider the circuit shown in Figure **??**. Suppose that we apply the test $AB = 00$. Let's also assume that $A/1$ is the only newly detected fault under this test vector. For this fault to be detected, the requirements are $A = 0$ and $C = 0$. The requirement $A = 0$ is already satisfied because A is a PI. For the requirement $C = 0$, the CRIPTR and the SVR Algorithms justify this requirement from line B since it is unreachable, resulting in the test $AB = 00$. However, this test can be relaxed to $AB = 0x$. To show this, assume that $AB = 0x$ and that stem A is faulty, i.e., $A = 0/1$. Thus, C will be $0/x$, and eventually $D = 0/1$, which means that $A/1$ is detected. Thus, in this example the assignment $C = 0$ is justified from a reachable line.

To take advantage from this situation, more involved rules need to be developed to determine the conditions under which a value can be justified from a reachable line. In this section, we develop such rules. The idea is to justify both the fault-free and the faulty machines. This

way, we don't have to worry about whether we choose to justify a value from a reachable or unreachable line. The only thing we have to worry about is that the fault-free and the faulty values are preserved and not masked. Of course, this has an implication, i.e., we have to compute the faulty value whenever we need it. The following example illustrates the idea.

Example 3.3:

Again, consider the circuit shown in Figure **??**. Assuming that the faulty values are already computed, we proceed as follows. To justify the fault-free 0 on line D, both fault-free 0s on lines C and $A1$ have to be justified . Thus, input A is marked as required. The fault-free 0 on line C is now justified from line A. The faulty value on line D is only justifiable from line $A1$ implying that the faulty value on line A is required as well. Thus, the fault-free and the faulty values on line D are justified only from line A, i.e., line B is relaxed resulting in the relaxed test $AB = 0x$.

Two-Values Relaxation (TVR) Algorithm

Algorithm **??** outlines the main part of the two-values relaxation (TVR) Algorithm. For every newly detected fault f under a test t, the Algorithm performs the following. First, it injects the fault f into the circuit and propagates it to an output O. While doing so, it also computes the faulty values. Every line l in the circuit is associated with two flags: l_{ffv} and l_{fv}. The first flag tells us whether the fault-free value of l is updated or not. The second one tells us the same thing about the faulty value of l. A flag is considered updated if it is updated at least once during one iteration of the test vectors loop in the main part of the TVR Algorithm. These flags tell us which value (faulty or fault-free value) of line l is required. The required value has its flag updated. So, in order to detect f, we update (set)

both counters of O. Then the next step is to justify the fault-free and the faulty machines by calling the event-driven procedure justify(O). After the justification of all the newly detected faults is over, we are sure that any primary input needed in the excitation or the propagation of the newly detected faults is marked. All unmarked primary inputs are turned into x's.

Algorithm 3.10 Main part of the TVR algorithm.

```
1:  for every test vector t do
2:     Fault simulate the circuit under the test t
3:     for every newly detected fault f do
4:        Inject f and propagate it to a primary output O
5:        Update O_ffv and O_fv
6:        justify(O)
7:     end for
8:     Output relaxed vector
9:     Mark all lines as non-required and reset all the flags
10: end for
```

The event-driven justification procedure of the faulty and the fault-free values is shown in Algorithm **??**. This procedure works as follows. After the event list E is initialized, we insert the output to be justified, O, into it. Then we do the following until E becomes empty. If l is a PI, the Algorithm simply marks it as required. Otherwise, if l_{ffv} is updated, then $justifyFFM(l)$ is called, and if l_{fv} is updated, then $justifyFM(l)$ is called.

The procedure $justifyFFM(l)$ is outlined in Algorithm **??**. The procedure checks whether l is a single-input, XOR, or XNOR gate, or whether the good value (fault-free value) of l, l_{gv} is a non-controlling value. If so, then all the fault-free flags of the inputs of l are updated and their associated lines are inserted into the event list E. However, if l_{gv} is a controlling value, then we check whether there is a controlling value input of l whose fault-free flag is updated. If such line does not exist, then cost functions are utilized to guide us to select the

controlling value input, j, whose cost is minimum. Then, the fault-free flag of line j, j_{ffv} is updated and j is inserted into the event list E.

The procedure $justifyFM(l)$ is outlined in Algorithm **??**. This procedure is exactly the same as $justifyFFM$ procedure, with one difference. The difference appears when we try to justify the faulty value of the line in which we excited the fault from. In such a case, the faulty value can not be justified; however, to ensure fault excitation, it is needed to justify the fault-free value. For this purpose, and at the beginning of this procedure, we check whether l is the point of excitation. If it is so, the fault-free flag of l is updated and l is inserted into the event list E. Otherwise, the faulty value of l is justified in the same manner as the justification of the fault-free value done in the procedure $justifyFFM$, but this time we consider the faulty value instead of the fault-free value.

The main improvement of TVR Algorithm over the SVR Algorithm is that one does not have to worry from where a controlling value input is justified as long as both the faulty and the fault-free values are justified. The overhead of the TVR Algorithm as compared to the SVR Algorithm is that we need to compute the faulty values until an output is reached. However, with this overhead, reachability analysis is unnecessary and is not done in the TVR Algorithm. Conceptually, performing reachability analysis of a fault in the SVR Algorithm is roughly equivalent to propagating that fault to a primary output and computing the faulty values in the way. Thus, the two Algorithms, roughly, have the same average CPU time, as will be shown in the experimental results in the next Chapter.

Algorithm 3.11 justify(O) - TVR algorithm version.

```
1: E ← ∅
2: Insert O into E
3: while E ≠ ∅ do
4:    l ← maximum level element of E
5:    Remove l from E
6:    if if l is a PI then
7:       mark l as required
8:    else
9:       if l_ffvc is updated then
10:         justifyFFM(l)
11:      end if
12:      if l_fvc is updated then
13:         justifyFM(l)
14:      end if
15:   end if
16: end while
```

Algorithm 3.12 justifyFFM(l)

```
1: if l is an output of a single-input, XOR, XNOR gate, or l_gv is a non-controlling value
   then
2:    for every input j of l do
3:       Update j_ffvc
4:       Insert j into E
5:    end for
6: else if no controlling value input of l is updated then
7:    j ← minimum cost input of l
8:    Update j_ffvc
9:    Insert j into E
10: end if
```

Algorithm 3.13 justifyFM(l)

```
1:  if l is the faulty line then
2:      Update l_ffvc
3:      Insert l into E
4:  else
5:      if l is an output of a single-input, XOR, XNOR gate, or l_fv is a non-controlling value
        then
6:          for every input j of l do
7:              Update j_fvc
8:              Insert j into E
9:          end for
10:     else if no controlling value input of l is updated then
11:         j ← minimum cost input of l
12:         Update j_fvc
13:         Insert j into E
14:     end if
15: end if
```

3.3.1 Worst-Case Analysis Of The TVR Algorithm

The same assumptions made for the worst-case analysis of the CRIPTR and the KMR techniques are assumed here as well.

For every newly detected fault f by a test vector t, the TVR technique justifies both the fault-free and the faulty machines. This requires at most $2N_F N_G$ basic operations. Thus, an upper-bound on the total number of basic operations is

$$2N_G N_F N_T \tag{3.3}$$

Note that this upper-bound is exactly the same upper-bound obtained for the SVR technique.

3.4 Selection Criteria

Note that in justifying a required controlling value, there could be several unreachable inputs with controlling value. In this case, priority is given to an input that is already marked as required. Otherwise, cost functions are used to guide the selection. Our objective from cost functions is to justify the required values by the smallest number of assignments on the primary inputs. This will result in increasing the number of x's extracted from relaxing a test vector. Cost functions are used to provide a relative measure on the selection that reduces the number of required assignments on the PIs.

The well-known recursive controllability cost functions [**?**] can be used for this purpose as they give a relative measure of the number of PI assignments required to justify a required value. For every line l, we compute two cost functions $C_0(l)$ and $C_1(l)$. For example, for an AND gate whose output is l and that has i inputs, the cost functions are computed as:

$$C_0(l) = \min_i C_0(i)$$

$$C_1(l) = \sum_i C_1(i)$$

The cost is computed for other gates in a similar way. Initially, $C_0(l)$ and $C_1(l)$ are assigned a value of 1 for PIs. In our work, we call these cost functions the regular cost functions. Regular cost functions are accurate for fanout-free circuits. However, when fanouts exist, regular cost functions do not take advantage of the fact that a stem can justify several required values.

To take advantage of that, we propose new cost functions, called the fanout-based cost functions. Note that these cost functions are different from the fanout-based cost functions

Figure 3.3: Illustration of selection criteria.

given in [**?**]. These functions are computed for an AND gate as follows. Let l be the output of an AND gate with i inputs. Let $F(l)$ denote the fanout (i.e., the number of fanout branches) of line l. Then, the fanout-based cost functions are computed as:

$$C_0(l) = \frac{\min_i C_0(i)}{F(l)}$$
$$C_1(l) = \frac{\sum_i C_1(i)}{F(l)}$$

The advantage of the proposed fanout-based cost functions over the regular cost functions given in [**?**] is illustrated by the following example.

Example 3.4:

Consider the circuit shown in Figure **??**. The detected faults under the shown test vector are $G6/0$ and $A/0$. The value 0 on $G5$ is required. Note that this value can be justified either through $G3$ or $G4$. If the regular cost functions, given in [**?**], are used then $C_0(G3) = 2$ and $C_0(G4) = 2$. If $G4$ is selected this will result in the two required values $E = 0$ and $F = 0$ and the test vector will be relaxed to $ABCDEF = 1xxx00$. If $G3$ is selected, then this may result in either of the following assignments $\{B = 0, C = 0\}$, $\{B = 0, D = 0\}$, $\{C = 0, D = 0\}$, or $\{C = 0\}$. According to this selection criterion, any of these choices is possible since they have the same cost. However, if the fanout-based criterion is used, then $C_0(G3) = 1$ and $C_0(G4) = 2$, which will select $G3$ to justify the value. Now in justifying $G1 = 0$, the assignment $C = 0$ will be selected since $C_0(C) = 1/2$ while $C_0(B) = 1$. Similarly, in justifying the requirement $G2 = 0$, the assignment $C = 0$ will be selected for the same reason. Thus, the test in this case will be relaxed to $1x0xxx$.

Figure 3.4: Another illustration of selection criteria.

While the fanout-based cost functions provide better selection criteria than the regular cost functions in general, there are some cases where this is not true as illustrated by the following example.

Example 3.5:

Consider the circuit shown in Figure **??**. The detected fault under the shown test vector is $G8/1$. To justify the required value on $G8$, we could either select $G7 = 0$ or $G = 0$. Using the fanout-based cost functions, $C_0(G7) = 1$ and $C_0(G) = 1$. If $G7 = 0$ is selected, then this will result in two primary input assignments, namely $B = 0$ and $E = 0$. However, using the regular cost functions $C_0(G7) = 2$ and $C_0(G) = 1$. Thus, the assignment $G = 0$ will be selected resulting in a more relaxed vector. Thus, in this example using the regular cost functions leads to a better solution.

To take advantage of both cost functions, we propose a weighted sum cost function of the two cost functions. Let $C_{01}(l)$ $(C_{11}(l))$ denote $C_0(l)$ $(C_1(l))$ based on the regular cost functions, while $C_{02}(l)$ $(C_{12}(l))$ denote $C_0(l)$ $(C_1(l))$ based on the fanout-based cost functions. Then, the proposed cost functions are as follows:

$$C_0(l) = A \cdot C_{01}(l) + B \cdot C_{02}(l) \tag{3.4}$$

$$C_1(l) = A \cdot C_{11}(l) + B \cdot C_{12}(l) \tag{3.5}$$

The weights A and B will be selected based on experimental results.

3.5 Theoretical Comparison with the KMR Technique

In this section, we compare our proposed techniques with the KMR technique. First, all the four techniques take a test set as input, and produce a partially specified test set as output. In KMR technique, the relaxed test set maintains the same fault coverage as the input test set. This is also the case for the SVR and the TVR techniques. However, in the CRIPTR technique, the fault coverage of the relaxed test set might drop a little bit, and it equals the fault coverage of the input test set as measured by using the critical path tracing Algorithm.

In KMR technique, a distinction is made between essential and nonessential faults. Test vectors are traversed one time to justify essential faults, then they are traversed another time to justify (unjustified) nonessential faults. After that, there might still be some undetected faults, even though these faults are treated explicitly. For this purpose, KMR technique traverses the test vectors for a third time, and uses extended justification to ensure the detection of the yet-undetected faults. However, in all our techniques, test vectors are traversed only once to justify all the faults detected by every vector. No distinction is made between essential and nonessential faults, and no faults are missed, except the faults missed due to the approximate nature of the CRIPT Algorithm in the CRIPTR technique. That is because we are employing reachability analysis in the CRIPTR and the SVR techniques, and because dealing with both fault-free and faulty values in the TVR technique.

Both the KMR technique and our techniques try to maximize the percentage of x's by propagating the fault under consideration through one primary output. However, unlike the KMR technique, our techniques utilize cost functions also to guide the justification of a controlling value in a way to maximize the percentage of x's.

As mentioned earlier, the order of test vectors affect the relaxation quality. However, dealing

with essential faults first , which is done in the KMR technique, might take care of the test vector ordering, and this might positively affect the relaxation quality obtained by the KMR technique.

In conclusion, the only advantage of the KMR technique over all our techniques is that they first justify essential faults. This, in a way, takes care of the test ordering problem to maximize the percentage of x's. However, the extended justification is considered as a primary disadvantage, as when applied, several x's are restored to their original values, and relaxation quality becomes very poor. Another disadvantage is that the test vectors are traversed three times.

In terms of upper-bounds on the number of basic operations, our SVR and TVR techniques are better than the KMR technique, and the saving is

$$2\left(3+2N_F\right)N_GN_T - 2N_GN_FN_T = 2(3+N_F)N_GN_T$$

This is an upper-bound on the overhead in terms of the number of basic operations when using the KMR Algorithm instead of using the SVR or the TVR Algorithms. However, in terms of complexity, the KMR, the SVR, and the TVR techniques have the same complexity, which is $O(N_GN_FN_T)$.

However, as compared to the CRIPTR Algorithm, the KMR Algorithm performs better, especially for large circuits with large number of fanout stems and large number of outputs. In such a situation, the complexity of the CRIPTR Algorithm tends to be cubic on the number of gates, in addition to the other two factors, i.e., the number of faults and the number of tests. In other words, for large circuits with large number of stems, where N_S

approaches N_G, and large number of outputs, where N_O approaches N_G, the complexity of the CRIPTR Algorithm becomes $O(N_T N_F N_G^3)$. However, under the same circumstances, the KMR Algorithm will have its complexity unaffected, because it is not a function of the number of stems and the number of outputs.

3.6 Concluding Remarks

In this chapter, we have described our proposed techniques. The first proposed technique is the CRIPTR technique. This technique is based on the critical path tracing Algorithm. The main disadvantage of this technique is that the fault coverage of the input test set might be reduced due to the approximate nature of the critical path tracing Algorithm. The second proposed technique is the SVR technique. This technique overcomes the disadvantage introduced by the CRIPTR technique and the fault coverage of the input test set is maintained. Both techniques, CRIPTR and SVR, suffer from a limitation. This limitation appears when we try to justify a reachable controlling value, and all controlling value inputs are reachable. In such a situation, the two techniques have no choice other than justifying all the inputs. However, the third proposed technique, the TVR technique, aims at removing this limitation. It does that by justifying both the faulty and the fault-free values of the observation output without worrying about reachable and unreachable lines. In other words, the main improvement of TVR Algorithm over the SVR Algorithm is that we don't have to worry from where we justify a controlling value input as long as both the faulty and the fault-free values are justified. The overhead of the TVR Algorithm as compared to the SVR Algorithm is that we need to compute the faulty values until an output is reached. However, reachability analysis is substituted by that and is unnecessary. Conceptually, performing

reachability analysis of a fault in the SVR Algorithm is roughly equivalent to propagating that fault to a primary output and computing the faulty values in the way. As will be shown in the experimental results in the next chapter, the two Algorithms in fact have almost the same average CPU time.

In this chapter also, we have proposed fanout-based cost functions. Cost functions, in general, guide us to select the minimum cost controlling value input when justifying a controlling value. Unlike the existing regular cost functions, the proposed fanout-based cost functions take advantage of the existence of fanouts in the circuit. To take advantage of both cost functions, i.e. the regular and the proposed fanout-based cost functions, we developed weighted sum of both cost functions. As will be shown in the next chapter, it is always better to use the weighted sum cost functions in all of the three proposed techniques. We also gave a theoretical comparison between our proposed techniques and the KMR technique. It was mentioned that the only advantage the KMR technique has is dealing with the essential faults first; however, this advantage is killed by the extended justification process.

Chapter 4

Experimental Results

In order to demonstrate the effectiveness of our proposed test relaxation techniques, we have performed experiments on a number of the largest ISCAS85 and full-scanned versions of ISCAS89 benchmark circuits shown in Table **??**. Before presenting our results, let us have a quick look at this table, which summarizes the characteristics of the ISCAS85 and ISCAS89 combinational and full-scan benchmark circuits. The first column gives the names of the benchmark circuits. Columns 2 to 5 indicate the number of inputs, outputs, gates, levels, collapsed faults based on equivalence, and test vectors applied, respectively. The used test sets are highly compacted and achieve 100% fault coverage of the detectable faults in each circuit. They are generated using the MinTest tool presented in [**?**]. It is clearly seen from the table how large the benchmarks we are dealing with in our experiments. The experiments were run on a SUN Ultra60 (UltraSparc II-450 MHZ) with a RAM of 512 MB. For fault simulation purposes, we have utilized HOPE fault simulator [**?**].

This Chapter is organized as follows. The next Section compares the best results of each

Circuit Name	No. Inputs	No. Outputs	No. Gates	No. Levels	No. Faults	No. Tests
c5315	178	123	2307	49	5350	37
c7552	207	108	3512	43	7550	73
c2670	233	140	1193	32	2747	44
s5378f	214	228	2779	25	4603	97
s9234f	247	250	5597	58	6927	105
s15850f	611	684	9772	82	11725	94
s13207f	700	790	7951	59	9815	233
s38584f	1464	1730	19253	56	36303	110
s38417f	1664	1742	22179	47	31180	68
s35932f	1763	2048	16065	29	39094	12

Table 4.1: Benchmark circuits characteristics.

one of our three proposed relaxation techniques with the BR method. The comparison is in terms of the percentage of x's and CPU time. We also compare the three solutions with each other. Then we experiment with cost functions in each one of the proposed solutions. After that, we present two applications of test relaxation. One application is related to test set compression. The other one is related to test compaction. As we will see, the two applications show how crucial it is to have an efficient test relaxation algorithm.

4.1 Comparison With the Results of the BR Method

In Table **??**, our proposed CRIPTR technique is compared with the BR method. The first column in the table indicate the circuit name. We compare the two techniques in terms of the fault coverage, the percentage of x's extracted, and the CPU time taken for relaxation. It is important to point out here that the fault coverage of the relaxed test set based on the BR method is the same as the fault coverage of the original test set, i.e. exact test set relaxation and no drop in the number of detected faults. However, the fault coverage of the relaxed test

	F.C.		% x's		Time (sec)	
Circuit	BR	CRIPTR	BR	CRIPTR	BR	CRIPTR
c5315	98.897	98.897	54.37	52.004	2376	2.0
c7552	98.265	98.119	55.45	52.154	1739	6.0
c2670	95.741	95.704	69.63	68.738	7008	1.0
s5378f	99.131	99.022	74.14	70.021	9312	3.0
s9234f	93.475	89.534	70.29	69.019	21945	6.0
s15850f	96.682	96.461	80.96	78.694	81122	20.0
s13207f	98.462	97.524	93.36	93.603	435420	28.0
s38584f	95.852	95.777	80.72	77.757	583000	94.0
s38417f	99.471	99.355	67.36	66.466	358156	60.0
s35932f	89.809	89.809	36.68	28.257	27180	22.0
	Average %x's		68.296	65.6713		

Table 4.2: Comparison between the proposed CRIPTR technique and the BR method ($A = 1$, $B = 6$).

set based on the CRIPTR technique may be reduced. This is due to the approximate nature of the critical path tracing Algorithm. The fault coverage of the relaxed test set based on the CRIPTR technique is equivalent to the fault coverage of the original test set as measured by the critical path tracing algorithm. As can be seen from the table, the drop in fault coverage is small (if any) for all the circuits.

It is interesting to observe that the CPU time taken by the CRIPTR technique is several orders of magnitude less than the BR method for most of the circuits. The BR method requires astronomical CPU times for large circuits and hence is impractical.

The percentage of x's obtained by the CRIPTR technique is also close to the percentage of x's obtained by the BR method for most of the circuits. The difference in the percentage of x's obtained ranges between -0.2% and 8.5%. For nine of the circuits out of ten, it is less than 4.2%. The average difference is 2.6%. Also note that we obtain higher percentage of x's for the circuit s13207f. These results are obtained using the proposed combined cost

	% x's		Time (sec)	
Circuit	BR	SVR	BR	SVR
c5315	54.37	52.141	2376	2.0
c7552	55.45	52.075	1739	8.0
c2670	69.63	68.767	7008	2.0
s5378f	74.14	70.898	9312	1.0
s9234f	70.29	66.416	21945	3.0
s15850f	80.96	78.833	81122	6.0
s13207f	93.36	92.928	435420	4.0
s38584f	80.72	77.981	583000	13.0
s38417f	67.36	66.227	358156	13.0
s35932f	36.68	28.238	27180	9.0
Average %x's	68.30	65.45		

Table 4.3: Comparison between the SVR technique and the BR method ($A = 1$, $B = 8$).

function, given in equations **??** and **??**, with $A = 1$ and $B = 6$. It will be shown next that this choice of A and B performs the best among other choices experimented with.

In Table **??**, the SVR technique is compared with the BR method in terms of percentage of x's extracted and CPU time taken for relaxation. It is important to point out here that the fault coverage of the relaxed test sets based on the BR method and the SVR technique is the same as the fault coverage of the original test set, i.e. exact test set relaxation and no drop in the fault coverage.

Again, it is observed that, for all the circuits, the CPU time taken by our proposed technique is less than the BR method by several orders of magnitude.

The percentage of x's obtained by the SVR technique is also close to the percentage of x's obtained by the BR method for most of the circuits. The difference in the percentage of x's obtained ranges between 0.4% and 8.4%. The average difference is 2.8%. For nine of the ten circuits, the difference is less than 4%. These results are obtained using $A = 1$ and $B = 8$ of the proposed combined cost function given in equations **??** and **??**.

	% x's		Time (sec)	
Circuit	BR	TVR	BR	TVR
c5315	54.37	52.141	2376	2.0
c7552	55.45	52.253	1739	6.9
c2670	69.63	68.787	7008	1.0
s5378f	74.14	71.100	9312	1.0
s9234f	70.29	66.509	21945	2.0
s15850f	80.96	78.871	81122	6.0
s13207f	93.36	92.929	435420	4.0
s38584f	80.72	78.070	583000	13.0
s38417f	67.36	66.255	358156	11.0
s35932f	36.68	28.238	27180	14.0
Average %x's	68.30	65.52		

Table 4.4: Comparison between the proposed TVR technique and the BR method ($A = 1$, $B = 16$).

Table **??** shows the results of the TVR algorithm as compared with the BR method. Comparable percentages of x's are obtained here as well, with very little time as compared to the BR method. However, for the circuit s35932f, the difference in the percentage of x's is still the same as in the SVR and the CRIPTR techniques. Unfortunately, the percentages of x's are almost identical to those of the SVR technique and only very little improvement has been achieved. The average improvement of the TVR over the SVR per circuit is 0.07%. This is obtained with $A = 1$ and $B = 16$. The only interpretation of that is to say that the situation discussed in the example given in Section **??**, where we can safely justify a controlling value from a reachable line, occurs seldom in these circuits.

Table **??** compares the CPU times and percentages of x's of the three proposed solutions. As clearly seen from the table, the CRIPTR technique has the best average relaxation, which is 65.67%. However, this only differs from the SVR technique in 0.22% and from the TVR technique in 0.15%. Most likely, the user will sacrifice that little increase in the percentage of x's to maintain the same fault coverage, i.e., either the SVR or the TVR

	CPU Time (Seconds)			Percentage of x's		
Circuit	CRIPTR	SVR	TVR	CRIPTR	SVR	TVR
c5315	2	2	2	52.00	52.14	52.14
c7552	6	8	7	52.15	52.08	52.25
c2670	1	2	1	68.74	68.77	68.79
s5378f	3	1	1	70.02	70.90	71.10
s9234f	6	3	2	69.02	66.42	66.51
s15850f	20	6	6	78.69	78.83	78.87
s13207f	28	4	4	93.60	92.93	92.93
s38584f	94	13	13	77.76	77.98	78.07
s38417f	60	13	11	66.47	66.23	66.26
s35932f	22	9	14	28.26	28.24	28.24
Average	24.2	6.1	6.1	65.67	65.45	65.52

Table 4.5: Comparison between the CPU times and the percentage of x's of three proposed solutions.

techniques will be preferred. However, the average CPU time difference between the SVR and the TVR techniques is 0 seconds. This indicates that the overhead CPU time consumed by the TVR technique as compared to the SVR technique is negligible, and thus TVR technique is preferable. However, the average difference between the CRIPTR and the TVR techniques is 18.1 seconds. This difference is due to the way we implement the procedure $Critical()$ shown in Algorithm **??**. This procedure forms the bottleneck of the CRIPTR technique. As mentioned earlier, in our implementation, checking a stem for criticality often involves fault simulation. This what makes the CRIPTR technique the slowest among our proposed techniques. However, Abramovici *et al.* [**?**] presented two efficient alternative implementations for the procedure $Critical()$. These implementations can be utilized to make the CRIPTR technique faster. It is mentioned in [**?**] that the critical path tracing algorithm can be implemented to be as fast as the concurrent fault simulators.

One might wonder and ask: why the BR method produces better relaxation quality than all our proposed techniques, although we are utilizing cost functions? The answer to this

question is obvious if we look at the implementation of the BR method. Suppose that we have a test set composed of the vectors $\{v_1, v_2, \ldots, v_n\}$. The BR method relaxes this set as follows. First, it takes the first bit of v_1, say b_{11}, and relaxes it, i.e., $b_{11} \leftarrow x$. Then, it fault-simulates the whole modified test set. Notice that most of the faults are detectable by a number of test vectors through a number of primary outputs. This is especially true for large circuits with highly compacted test sets, which is our case. So, the chance that b_{11} is relaxed and the fault coverage remains the same is somehow high. However, in our techniques, we are only aware of faults that are detected by the previously processed test vectors. Also, we don't control the selection of the output. The selection of the output depends on the order in which primary outputs are traversed in the CRIPTR technique. In the SVR and TVR techniques, the selection of the output to be justified depends on which output is reached first in the reachability analysis phase. So, the BR method has a global view of the test set and the outputs, and this what makes it superior in the percentage of x's extracted. In our techniques, further research can be conducted to propagate the faults through an output which produces better relaxation quality. Also, one might think of reordering the test vectors to obtain better quality solutions.

4.2 Cost Function Experiments

A large number of experiments have been performed to examine how the proposed cost functions given in Equations **??** and **??** affect the relaxation quality of the three proposed techniques. In each experiment, we fix the weight parameters, A and B, to each one of the values given in the columns of Table **??**. Note that weight A is for the regular cost function and weight B is for the fanout-based cost function. This is done for all the ten

A	0	0	1	1	1	1	1	1	1	1	1	1	1	1
B	0	1	0	1	2	3	4	5	6	7	8	16	32	64

Table 4.6: Weight combinations experimented with.

benchmark circuits given in Table **??** for every one of the proposed techniques. Thus, a total of $14 \times 10 \times 3 = 420$ experiments have been performed. A perl script has been written to help in the automation of performing this large number of experiments. Although the number of experiments is large, the total execution time of all the experiments did not exceed 85 minutes. This proves that our proposed techniques are very fast as compared to the BR method, which might take astronomical CPU time to complete the same number of experiments. Tables **??**, **??**, and **??** show only a subset of the results obtained for the CRIPTR, SVR, and TVR techniques, respectively. As can be seen from these tables, for all the circuits, the use of cost functions results in higher percentage of x's extracted than without using the cost functions. A difference of up to 5% is observed. Furthermore, it is clearly indicated from the tables that the proposed fanout-based cost function produces better results than the regular cost function. However, for the CRIPTR technique, a combined cost function with a weight of 1 for the regular cost function and a weight of 6 for the fanout-based cost function seems to be a good cost measure as it provides the highest percentage of extracted x's on average. For the SVR technique, a combined cost function with a weight of 1 for the regular cost function and a weight of 8 for the fanout-based cost function seems to be a good heuristic. Finally, a combined cost function with a weight of 1 for the regular cost function and a weight of 16 for the fanout-based cost function produces the best result for the TVR technique. In each one of the three tables, the maximum percentage of x's obtained for every circuit is highlighted.

Circuit	A=0 B=0	A=0 B=1	A=1 B=0	A=1 B=1	A=1 B=4	A=1 B=6	A=1 B=8	A=1 B=16
c5315	48.892	51.989	49.985	52.004	52.004	52.004	**52.156**	52.141
c7552	48.230	52.174	48.415	52.154	52.154	52.154	52.154	**52.253**
c2670	66.026	68.357	66.631	68.718	68.728	68.738	**68.787**	**68.787**
s5378f	67.641	70.209	69.140	69.756	69.920	70.021	70.927	**71.100**
s9234f	66.686	68.571	67.446	68.845	68.857	**69.019**	66.497	66.509
s15850f	77.682	78.866	77.849	78.292	78.689	78.694	78.856	**78.871**
s13207f	93.112	93.508	93.235	93.597	**93.604**	93.603	92.931	92.929
s38584f	75.418	77.838	75.930	77.661	77.705	77.757	77.927	**78.070**
s38417f	65.834	**66.655**	66.201	66.487	66.466	66.466	66.231	66.261
s35932f	23.057	27.444	28.172	**28.257**	**28.257**	**28.257**	28.238	28.238
Average	63.258	65.561	64.300	65.577	65.638	**65.671**	65.470	65.516

Table 4.7: Effect of cost function on the extracted percentage of x's using the CRIPTR Algorithm.

Circuit	A=0 B=0	A=0 B=1	A=1 B=0	A=1 B=1	A=1 B=4	A=1 B=6	A=1 B=8	A=1 B=16
c5315	48.527	52.065	50.076	52.080	52.080	52.080	**52.141**	**52.141**
c7552	48.091	52.068	48.329	52.075	52.075	52.075	52.075	**52.174**
c2670	65.899	68.377	66.407	68.748	**68.767**	**68.767**	**68.767**	**68.767**
s5378f	68.422	**71.057**	69.891	70.484	70.652	70.753	70.898	**71.057**
s9234f	63.621	65.949	64.372	66.046	66.189	66.408	**66.416**	65.980
s15850f	77.693	**78.971**	77.855	78.391	78.823	78.830	78.833	78.847
s13207f	92.457	92.920	92.485	92.920	**92.928**	**92.928**	**92.928**	**92.928**
s38584f	75.328	**78.072**	75.839	77.794	77.827	77.951	77.981	**78.072**
s38417f	65.518	**66.465**	65.865	66.162	66.171	66.171	66.227	66.247
s35932f	22.986	27.415	28.120	**28.238**	**28.238**	**28.238**	**28.238**	**28.238**
Average	62.854	65.336	63.924	65.294	65.375	65.420	**65.450**	65.445

Table 4.8: Effect of cost function on the extracted percentage of x's using the SVR Algorithm.

Circuit	A=0 B=0	A=0 B=1	A=1 B=0	A=1 B=1	A=1 B=4	A=1 B=6	A=1 B=8	A=1 B=16
c5315	47.616	52.035	49.484	52.095	52.095	52.095	**52.156**	52.141
c7552	47.773	52.181	47.806	52.154	52.154	52.154	52.154	**52.253**
c2670	65.178	68.338	65.977	68.728	68.738	**68.787**	**68.787**	**68.787**
s5378f	68.166	**71.115**	69.636	70.445	70.619	70.758	70.927	71.100
s9234f	62.768	66.493	63.686	66.100	66.281	66.489	66.497	**66.509**
s15850f	77.484	**79.030**	77.660	78.351	78.833	78.852	78.856	78.871
s13207f	91.957	92.922	91.990	92.920	**92.931**	**92.931**	**92.931**	92.929
s38584f	74.803	**78.081**	75.355	77.722	77.760	77.885	77.927	78.070
s38417f	65.393	**66.458**	65.768	66.146	66.172	66.172	66.231	66.261
s35932f	22.882	27.430	28.096	**28.238**	**28.238**	**28.238**	**28.238**	**28.238**
Average	62.402	65.408	63.546	65.290	65.382	65.436	65.470	**65.516**

Table 4.9: Effect of cost function on the extracted percentage of x's using the TVR algorithm.

4.3 Example Applications of Test Relaxation

To illustrate the application of test relaxation in improving the effectiveness of test compression, we have applied the Frequency-Directed Run-Length (FDR) compression technique, Extended Frequency-Directed Run-Length (EFDR) compression technique, and compression technique based on geometric shapes (GPB), which are all discussed in Chapter 2. For the GPB technique, blocks of size 8×8 are used with 01-distance sorting Algorithm. We apply these compression techniques on the used test sets. Tables **??**, **??**, and **??** show the test compression results for the FDR, the EFDR, and the GPB compression techniques, respectively. In all these tables, the first column shows the circuit name, and the last four columns show the compression ratio of four test sets. One is the original test set without relaxation. Another one is the relaxed test set based on the BR method. The third ratio is for the relaxed test set based on the proposed SVR technique, and the fourth is for the relaxed test set based on the proposed TVR technique. As shown in these tables, for all the circuits,

Circuit	Compression Ratio			
	Fully Specified Tests	Relaxed Tests		
		BR	SVR	TVR
c5315	-25.63	20.47	20.10	20.04
c7552	-19.48	37.13	34.17	34.22
c2670	-24.93	43.84	43.46	43.46
s5378	-27.86	46.85	44.10	44.41
s9234.1	-25.52	34.80	32.01	32.16
s15850.1	-23.41	56.49	56.77	56.88
s13207.1	-25.97	78.78	79.51	79.51
s38417	-29.08	37.66	35.99	36.08

Table 4.10: Effect of relaxation on test compression ratio using the FDR codes.

all compression techniques used achieve negative compression (i.e., compressed test set is larger than original test set) on the original (fully-specified) test sets. However, significant compression is achieved based on the relaxed test sets. All the BR method, SVR technique, and TVR technique achieve comparable compression ratio for most of the circuits. For the circuit c5315, and under the EFDR technique, a difference of about 6% is observed between the compression ratios of the test set relaxed by the SVR technique and the test set relaxed by the TVR technique. Notice that both SVR and TVR techniques obtain the same relaxation quality for this circuit as shown in Table **??**. In general, the higher the percentage of x's extracted, the higher the compression ratio. However, the location of x's and their distribution certainly have an impact on the results. From these results, it is clear that in order to have effective test compression, it is crucial to have a relaxed test set and an efficient test relaxation technique.

In order to demonstrate the impact of test relaxation on test compaction, we used HITEC [**?**] to generate test sets that detect all the detectable faults on some of the benchmark circuits used. The results are shown only for six of the total ten benchmarks, because, for the remaining four circuits, HITEC was unable to generate test sets which fully cover all de-

Circuit	Compression Ratio			
	Fully Specified Tests	Relaxed Tests		
		BR	SVR	TVR
c5315	-16.91	28.66	26.13	20.08
c7552	-11.17	41.48	38.95	39.12
c2670	-15.16	53.10	52.10	52.05
s5378	-17.72	50.81	47.65	47.97
s9234.1	-16.68	37.82	34.80	35.01
s15850.1	-17.20	56.29	56.48	56.44
s13207.1	-17.85	79.38	79.98	79.95
s38417	-7.26	52.35	48.44	48.47

Table 4.11: Effect of relaxation on test compression ratio using the EFDR codes.

Circuit	Compression Ratio			
	Fully Specified Tests	Relaxed Tests		
		BR	SVR	TVR
c5315	-4.10	27.88	24.14	24.51
c7552	-2.68	37.75	34.92	34.90
c2670	-3.50	51.85	48.93	48.91
s5378f	-3.57	51.55	48.64	48.88
s9234f	-3.48	43.45	38.59	39.08
s15850f	-3.28	60.32	57.24	57.05
s13207f	-3.25	84.14	83.65	83.62
s38417f	-2.70	46.50	43.11	43.11

Table 4.12: Effect of relaxation on test compression ratio using the GPB compression.

Circuit	Number of Test Vectors				
	Original	ROFS	BR	SVR	TVR
c5315	193	119	100	103	104
c2670	154	106	98	101	103
s5378	359	252	136	140	138
s9234.1	620	376	200	214	215
s13207.1	633	478	252	257	257
s35932	80	63	37	33	33

Table 4.13: Effect of relaxation on test compaction.

tectable faults during the specified time limit. The second column in Table **??** shows the number of test vectors obtained by HITEC. Then, we compacted the test vectors based the reverse-order and random order fault simulation for 20 iterations. The number of the compacted test vectors is shown in the third column of the table. Next, we relaxed the compacted test sets based on both the BR, SVR, and TVR techniques, respectively. To achieve further compaction on the relaxed test sets, we merged compatible test vectors. The fourth, fifth, and sixth columns show the number of merged test vectors based on the BR relaxed test set, the relaxed test set based on the SVR technique, and the relaxed test set based on the TVR technique, respectively. As can be seen from the results, over 40% test compaction is achieved for most of the circuits based on merging compatible vectors. Due to the larger percentage of x's achieved by the BR method, more compacted test sets are obtained except for circuit s35932. As demonstrated by the results, starting with a compacted test set, significant further compaction can be achieved by relaxing the test set and merging compatible vectors.

4.4 Concluding Remarks

In this Chapter, the effectiveness of our proposed techniques has been demonstrated by comparing them with the BR method. Although the BR method obtains slightly higher average percentage of x's than our techniques, all our techniques are several orders of magnitude faster than the BR method.

The first proposed technique was the CRIPTR technique. This technique suffers from the minor drop in fault coverage of the relaxed test set due to the use of the critical path tracing algorithm. However, in the SVR technique, we overcome this problem by maintaining the same fault coverage as the fully specified test set. The problem of justifying a controlling value from a reachable line is solved by the TVR technique. In this technique, we don't worry from where we justify the controlling value as long as we justify both the faulty and the fault-free machines. The improvement due to this technique over the SVR technique is only minor. However, the average CPU overhead time is almost zero, which means that they (the SVR and the TVR techniques) both obtain the result within the same time. Due to this fact, we recommend using the TVR technique.

In this chapter also, we have shown how cost functions affect the quality of relaxation. Several experiments have been performed to show this effect. We also demonstrated the usefulness of our technique to both compression and compaction. From the results obtained, it is clear that in order to have effective test compression and compaction, it is crucial to have a relaxed test set and an efficient test relaxation technique.

Chapter 5

Conclusions And Future Research

In this work, we have presented three efficient test relaxation techniques for combinational circuits. All the techniques are faster than the BR method by several orders of magnitude. The first technique, i.e. the CRIPTR technique, is based on the critical path tracing Algorithm, and hence may result in a small drop in the fault coverage after relaxation. This is due to the approximate nature of the CRIPT Algorithm. However, based on experimental results, only a small drop in the fault coverage is observed for most of the circuits.

In the second proposed technique, i.e. the SVR technique, the relaxed test set maintains the same fault coverage of the original test set. This technique simply identifies all the newly detected faults under a given test vector and marks all the lines whose values are required to detect these faults. Any unmarked input is turned into an x. The CRIPTR and the SVR techniques avoid justifying a controlling value from a reachable line.

The third solution, i.e. the TVR technique, is an improvement over the SVR technique. It tries to justify both the fault-free and the faulty machines without worrying from which line

a controlling value is justified. As we saw in the experimental results, only little improvement is achieved without any overhead time.

Furthermore, the percentage of x's extracted is close to the one obtained by the BR method. The CRIPTR technique has the best average relaxation, which is 65.67%. However, this only differs from the SVR technique in 0.22% and from the TVR technique in 0.15%. Most likely, we would sacrifice that little increase in the percentage of x's to maintain the same fault coverage, i.e., in most cases, we favor either the SVR or the TVR techniques because they maintain the fault coverage of the original test set.

Having an efficient test relaxation technique is crucial for effective test compaction and compression. Application of test relaxation in achieving more effective test compaction and compression has been demonstrated.

Finally, let us shed some light on the future directions of this work. Further research can be conducted to control the propagation of the faults through an output to obtain better relaxation quality. Currently, our proposed Algorithms propagate a fault to an output, not necessarily the best output. This might be possible with the help of modified versions of observability functions found in [**?**] .

Also, it has been observed that the order in which test vectors are traversed affects the relaxation quality. Thus, one might think of test vectors reordering, and studies its effect on the quality of relaxation. A simple rule is that a vector which detects a large number of faults should be traversed earlier than a vector which detects only small number of faults.

Also, one might think of developing better cost functions for both the justification of a controlling value and the selection of the best propagation output. A final idea is to improve

the techniques in order to have some sort of parallelism, i.e., try to process several faults at one iteration.

Vita

- Ali Saleh Al-Suwaiyan
- Born in Unaizah, Saudi Arabia on July 3, 1976
- Received B.S. degree in Computer Engineering from KFUPM, Saudi Arabia in February 1998
- Completed M.S. degree requirements at KFUPM, Saudi Arabia in October 2002