An Efficient Test Relaxation Technique for Synchronous Sequential Circuits

Aiman El-Maleh and Khaled Al-Utaibi
King Fahd University of Petroleum and Minerals
Dhahran 31261, Saudi Arabia
emails:{aimane, alutaibi}@ccse.kfupm.edu.sa

Abstract

Testing systems-on-a-chip (SOC) involves applying huge amounts of test data, which is stored in the tester memory and then transferred to the circuit under test (CUT) during test application. Therefore, practical techniques, such as test compression and compaction, are required to reduce the amount of test data in order to reduce both the total testing time and the memory requirements for the tester. Relaxing test sequences can improve the efficiency of both test compression and test compaction. In addition, the relaxation process can identify self-initializing test sequences for synchronous sequential circuits. In this paper, we propose an efficient test relaxation technique for synchronous sequential circuits that maximizes the number of unspecified bits while maintaining the same fault coverage as the original test set.

1 Introduction

Rapid advancement in VLSI technology has lead to a new paradigm in designing integrated circuits where a system-on-a-chip (SOC) is constructed based on pre-designed and pre-verified cores such as CPUs, digital signal processors, and RAMs. Testing these cores requires a large a mount of test data which is continuously increasing with the rapid increase in the complexity of SOC. This has a direct impact on the total testing time and the memory requirements of the testing equipment. Hence, reducing the amount of test data is considered as one of the challenging problems in testing.

Test compression and compaction techniques are widely used to reduce the storage and test time by reducing the size of the test data. Test compression techniques can achieve better results if the test set is composed of test cubes (i.e. if the test set is partially specified). In fact, some compression techniques such as, LFSR-reseeding [1, 2], require the test vectors to be partially specified. Even those techniques which require fully specified test data can benefit from unspecified bits in the test set. For example, variable-to-fixed-length coding [3] and variable-to-variable-length coding [4, 5] are known to perform better for long runs of 0's. Hence, assigning 0's to the don't care values in the test set will improve the efficiency of these techniques. Similarly, run-length coding techniques [6] can specify the don't care values in a way that will reduce test vector activity (i.e. the number of transitions from 0 to 1 and vice versa), which in tern improves the compression efficiency.

Test compaction techniques can also benefit from a partially specified test. For example, when merging two test sequences using the overlapping compaction techniques described in [7], a don't care value, 'X', can be merged with any one of the values: '0', '1', and 'X'. Therefore, increasing the number of X's in a test set will reduce the number of conflicts that may occur while merging two test sequences, and hence, improves the efficiency of the compaction process.

Test-relaxation can also identify self-initializing test sequences in synchronous sequential circuits. During the test-relaxation process, if a memory-element is not required to justify any one of the detected faults, then it can be relaxed (i.e set to an 'X'). Then, any time frame with all memory-elements set to X's is considered as the start of a new self-initializing test sequence.

In this paper, we propose an efficient test relaxation technique for synchronous sequential circuits that maximizes the number of unspecified bits while maintaining the same fault coverage as the original test set. The rest of this paper is organized as follows. Section 2 defines the targeted problem, and summarizes previous work. Section 3 illustrates our idea with examples. Section 4 formally describes our test relaxation algorithm. Section 5 describes the selection criteria used by our technique. Experimental results are given in section 6, and section 7 concludes the paper.

2 Problem Definition

The problem of test relaxation for synchronous sequential circuits, i.e. extracting a partially specified test set from a fully-specified one, has not been solved effectively in the literature. This problem, which is targeted in this paper, can be defined as follows. *Given a synchronous sequential circuit and a fully specified test set, generate a partially*

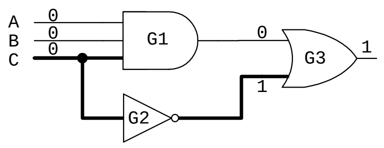

Figure 1: Circuit of Example 1.

specified test set that maintains the same fault coverage as the fully specified one while maximizing the number of unspecified bits. One obvious way to solve this problem is to use a bitwise-relaxation technique, where we test for every bit in the test set whether changing it to an 'X' reduces the fault coverage or not. Obviously, this technique is $O(nm)$ fault simulation runs, where n is the width of one test vector, and m is the number of test vectors. Obviously, this technique is impractical for large circuits. A partially specified test set can also be obtained using dynamic ATPG compaction. In dynamic compaction, every test vector is processed immediately after its generation in order to specify unspecified primary inputs (PIs). This feature can be disabled to obtain a compact and relaxed test set. However, this technique does not solve the problem of relaxing an already existing test set. In addition, this technique cannot benefit from random test pattern generation, because it is fault oriented.

Recently, two test relaxation techniques for combinational and full-scan sequential circuits were proposed in [8, 9]. The main idea of both techniques is to determine logic values in the fully-specified test set that are necessary to cover (i.e. detect) all faults which are detectable by this test set. Unnecessary logic values are set to X's. However, as far as synchronous sequential circuits are concerned, the only existing solution to the problem of relaxing a given test set is the bitwise-relaxation method.

3 Illustrative Examples

The techniques proposed in [8, 9] justify detected faults based on logical values only, which may result in fault masking. Each one of these techniques handles this problem in a different manner. In the first technique, when a fault effect is propagated to at least one input of a gate, then all inputs of this gate are justified. The second technique, on the other hand, uses some rules based on fault-reachability analysis to avoid fault masking. The two approaches are illustrated in the following example.

Example 1: Consider the circuit shown in Figure 1 under the fault $C/1$. The assignments $C = 0$ and $G1 = 0$ are necessary to excite this fault and propagate it to the primary-output. The assignment $G1 = 0$ can be satisfied by any one of the assignments $A = 0$, $B = 0$ or $C = 0$. However, selecting the assignment $C = 0$ and relaxing the two inputs A and B will result in masking the fault $C/1$. To avoid fault masking, the first technique justifies all the inputs of $G1$, while the second technique justifies the the assignment on $G1$ through an input that is not reachable from the fault $C/1$. In this case, either one of the two assignments $A = 0$ or $B = 0$ can be selected.

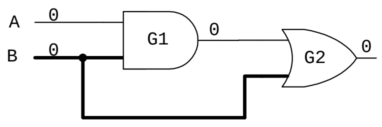

Figure 2: Circuit of Example 2.

It is clear that the second technique results in more relaxation. However, there are some situations, as will be shown in the next example, where it is possible to justify the required value from a reachable line without masking any of the detected faults.

Example 2: Consider the circuit shown in Figure 2 under the fault $B/1$. The assignments $B = 0$ and $G1 = 0$ are required to excite and propagate this fault to the primary-output. According to the second technique, the assignment $G1 = 0$ can't be justified through B because this input is reachable from the fault $B/1$. However, if we relax A ($A = X$), we find that the fault is still detected (i.e. the fault has not been masked). The reason is that the faulty value of B (i.e. 1) is considered as a controlling value for the gate $G2$. Thus, the fault $B/1$ will propagate to primary-output regardless of the faulty-value propagating through $G1$. This limitation can be avoided if we consider both fault-free and faulty values of the circuit when justifying a given fault. In Figure 2, the circuit lines have the following combinations of fault-free/faulty values: $A = 0/0$, $B = 0/1$, $G1 = 0/0$ and $G2 = 0/1$. In order to justify the fault $B/1$, it is enough to justify the fault-free/faulty values on the primary-output $G2$. So, the assignment $G2 = 0/1$ can be satisfied by the assignments $B = 0/1$ and $G1 = 0/X$. Notice that the faulty-value of $G1$ is not required (i.e. set to 'X'). Hence, either one of the two inputs A or B can satisfy this assignment. Since B has been selected to satisfy the assignment on the primary-output, it can be selected as well to satisfy the assignment on $G1$. In this case, we can relax A.

Our proposed technique extends the fault-free/faulty values justification to handle synchronous sequential circuits. A synchronous sequential circuit can be represented as a linear iterative array of combinational cells. Each cell represents one time frame in which the current states of the flip-flops become pseudo-inputs (y_i), and the next states become pseudo-outputs (Y_i). So, we need to fault simulate the circuit under the given test set to determine faults detected in every time frame. Then, we can justify fault-

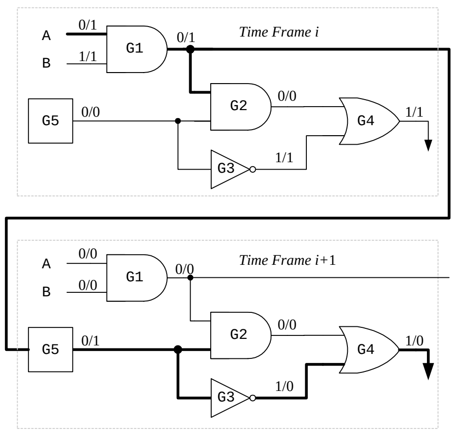

Figure 3: Circuit of Example 3.

free/faulty values necessary to detected these faults starting from the last time frame backwards. The justification process is performed frame by frame. In every time frame i, all fault-free/faulty values necessary to detect a newly detected fault are justified starting from primary-outputs towards primary-inputs and/or memory-elements. If the fault-free/faulty value of a primary-input is required to justify any one of the newly detected faults, then the value of this primary-input should be specified in the test set. On the other hand, required values on the memory-elements need to be justified when the next time frame, $i-1$, is processed. The general behavior of the proposed technique is illustrated by the following example.

Example 3: Consider the iterative-array-model shown in Figure 3. This model represents two time frames of a synchronous sequential circuit under two test vectors: $t_i = 01$ and $t_{i+1} = 00$. Assume that the only newly detected fault is $A/1$, i.e., other faults are either previously detected by earlier test vectors, or not part of the fault list. In t_i, the fault $A/1$ is excited and propagated to the memory element $G5$, but is not yet detected. In t_{i+1}, the fault $A/1$ is propagated to the primary-output, $G4$, where it gets detected. So, in order to justify this fault, it is enough to justify the fault-free/faulty values on the primary-output, $G4$, in t_{i+1}. The assignment $G4 = 1/0$ can be satisfied by the two assignments $G2 = X/0$ and $G3 = 1/0$. Next, we justify the assignment $G2 = X/0$. This can be satisfied by the assignment $G1 = X/0$, which in turn can be satisfied by either one of the two inputs A or B (i.e one of these inputs can be relaxed). The assignment $G3 = 1/0$, on the other hand, needs to be justified through $G5$. Since $G5$ is a memory-element, its fault-free/faulty values should be justified in the previous time frame (t_i). Therefore, we need to justify the faulty-free/faulty values of $G5$ in time frame t_i. These values can be satisfied by the two assignments $A = 0/1$ and $B = 1/1$. Since the fault-free/faulty values of both inputs are required, none of these inputs can be relaxed in this time frame.

4 Proposed Technique

Before describing the proposed technique, we give the following definitions.

Definition 1 *The good value of a gate g, denoted by GoodValue(g), is the value of the gate under the fault-free machine.*

Definition 2 *The faulty value of a gate g, denoted by FaultyValue(g), is the value of the gate under the faulty machine.*

Definition 3 *The justify value of a gate g, denoted by JustifyValue(g), is the fault-free/faulty assignment that needs to be justified by g.*

Due to the nature of sequential circuits (i.e. feedback from memory-elements), a fault excited in one time frame might propagate through several time farms before it gets detected. Hence, several time frames need to be traced back to justify such faults. Therefore, we need to store enough information about fault propagation, detection and justification in order to perform the justification process frame by frame. Five lists are used to store the the required information: *POJustificationList*, *PPOJustificationList*, *MEJustificationList*, *FaultPropagationList*, and *EventList*. The purpose of each one of these four lists is explained below.

The purpose of the *POJustificationList* is to store newly detected faults in every time frame. These faults will be justified backwards starting from the time frames where they first get detected. During fault simulation, if a fault f propagates to one or more memory-elements, then these memory-elements and their faulty values are added to the *MEPropagationList*. The *PPOJustificationList* is used to store faults that can't be completely justified during a certain time frame. Notice that if one or more memory-elements are required to justify a fault f during some time frame i, then f can't be completely justified during this time frame. Hence, these memory-elements will be added to *MEJustificationList*[f], and the justification of f will continue during time frame $i-1$. The *EventList* keeps track of the gates that need to be justified for a certain fault. Gates are inserted in event list according to their levels in the circuit.

Algorithm 1 shows an outline of the proposed test relaxation technique which consists of three phases. The first phase initializes the five lists. The *RelaxedTestSet*, as the name indicates, represents the relaxed test set. Initially, all the bits in this set are X's. However, more bits will be specified throughout the relaxation process in order to justify the detected faults.

Fault simulation is performed in the second phase to identify newly detected faults. These faults are stored in *POJustificationList*[i] for every test vector i. During fault simulation, if a fault f propagates to one or more memory-elements, then these memory-elements together with their faulty values are added to *FaultPropagationList*[f]. The information in this list will be used to compute faulty values of the circuit during the justification phase. It is important to point out here that we need to store the logical values of the memory-elements for all the time frames. This will enable the third phase to perform logic simulation in a certain time frame independent of the other time frames.

The third phase starts from the last time frame down to the first one. In every time frame, i, the algorithm performs the following. First, it logic simulates the circuit under the test vector i to determine the good value of every gate. Then, it checks *PPOJustificationList*[i] for any fault that has not been completely justified in time frame $i+1$. Unjustified faults are removed from the list and justified one by one. Next, it checks *POJustificationList*[i] for newly detected faults and justifies them. Justifying a fault, f, involves two operations: (1) Computing the faulty-values of the circuit under the fault f and (2) Backward justification. These operations are described bellow.

Local fault simulation is used to compute the faulty-values of the circuit under a given fault f. The process starts by injecting the fault f at its corresponding line in the circuit. Then, it sets the faulty-values of the memory-elements according to the faulty-values propagating from time frame $i-1$. Next, the fault effects on the faulty-line and memory-elements are forward propagated to determine the faulty-values of all gates in the circuit.

Algorithm 2 shows the justification process of a fault f in time frame t. In this algorithm, the event list is processed level by level starting from the maximum level. In each level, the the required values on a gate g (i.e. *JustifyValue(g)*) are satisfied according to the following procedure. First, the algorithm determines the corresponding values (v_1/v_2) on the input(s) of the gate g. For example, if the required values on the output of an *inverter* are $0/1$, then the corresponding requirements on the input of this gate are $1/0$. The next step is to justify v_1/v_2 through the input(s) of g as follows.

If g is a primary-input (*PI*), then we need to specify its value whenever the required fault-free/faulty value is not 'X'.

A requirement on a memory-element (*DFF*) can't be justified in the current time frame (i). Therefore, the memory-element is added to *MEJustificationList*[f], and the fault f is added to the justification list of the next time frame ($i-1$).

Algorithm 1 Main Algorithm

```
(*Initialization phase*)
for every fault, $f$, in the fault list of the given circuit do
    Let FaultPropagationList[$f$] $\leftarrow \phi$
    Let MEJustificationList[$f$] $\leftarrow \phi$
for every test vector $i$ do
    Let POJustificationList[$i$] $\leftarrow \phi$
    Let PPOJustificationList[$i$] $\leftarrow \phi$
    for every level, $l$, of the given circuit do
        Let EventList[$l$] $\leftarrow \phi$
    for every primary input j do
        Let RelaxedTestSet[$i$][$j$] $\leftarrow$ 'X'
(*Fault simulation phase*)
for $i \leftarrow 1$ to $n$ do
    Fault simulate the circuit under test vector $i$
    for every fault, $f$, newly detected in $i$ do
        Add $f$ to POJustificationList[$i$]
    for every fault $f$ propagating to time frame $i+1$ do
        Add all memory-elements reachable from the fault
        $f$ together with their faulty values to FaultPropaga-
        tionList[$f$]
(*Fault justification phase*)
for $i \leftarrow n$ downto 1 do
    Logic simulate the circuit under the test vector $i$
    while PPOJustificationList[$i$] $\neq \phi$ do
        Remove $f$ from PPOJustificationList[$i$]
        Compute faulty values of the circuit under the fault
        $f$
        for every memory-element, $d$, whose fault-
        free/faulty values are required to justify $f$ in time
        frame $i+1$ do
            Remove $d$ from MEJustificationList[$f$]
            Let $j$ be the input of $d$
            Add $j$ to EventList[level($j$)]
        Justify($f$, $i$)
    while POJustificationList[$i$] $\neq \phi$ do
        Remove $f$ from POJustificationList[level($j$)]
        Compute faulty values of the circuit under the fault
        $f$
        Let $j$ be a primary-output at which the fault $f$ gets
        detected
        Add $j$ to EventList[level($j$)]
        Justify($f$, $i$)
```

If g is an inverter (*NOT*) or a buffer (*BUF*), then its input is required to justify v_1/v_2. Hence, the input of g is added to the proper level in the event list. If the fault-free/faulty value of an *XOR* or *XNOR* gate is required, then the fault-free/faulty values on every input of the gate are required as well.

If g is an *AND*, *OR*, *NAND* or *NOR* gate, then we have four different possibilities. First, both v_1 and v_2 are controlling values of g. In this case, the algorithm searches for an input that satisfies both values and adds it to the event

Algorithm 2 Justify(f, i)

```
for every level, $l$, of the circuit do
    while EventList[$l$]$\neq \phi$ do
        Remove gate $g$ from the EventList[$l$]
        Let ($v_1$,$v_2$)$\leftarrow$JustifyValue($g$)
        if $g$ is (NOT|NAND|NOR) then
            Let ($v_1$,$v_2$) $\leftarrow$ ($\bar{v_1}$,$\bar{v_2}$)
        case g is
            (1) PI:
                if $v_1$ $\neq$'X' then
                    Let RelaxedTestSet[$i$][$g$]$\leftarrow$ $v_1$
                else if $v_2$ $\neq$'X' then
                    Let RelaxedTestSet[$i$][$g$]$\leftarrow$ $v_2$
            (2) DFF:
                Add $f$ to PPOJustificationList[$i-1$]
                Add $g$ to MEJustificationList[$f$]
            (3) BUF|NOT:
                Let $j$ be the input of $g$
                Let JustifyValue($j$)$\leftarrow$($v_1$, $v_2$)
                Add $j$ to EventList[level($j$)]
            (4) XOR|XNOR:
                for every input, $j$, of $g$ do
                    Let $v_1$ $\leftarrow$ GoodValue($j$)
                    Let $v_2$ $\leftarrow$ FaultyValue($j$)
                    Let Justifyvalue($j$) $\leftarrow$ ($v_1$, $v_2$)
                    Add $j$ to EventList[level($j$)]
            (5) AND|OR|NAND|NOR:
                if $v_1$ and $v_2$ are controlling values of $g$ then
                    Find an input, $j$, of $g$ that satisfy $v_1$
                    Find an input, $k$, of $g$ that satisfy $v_2$
                    if $j$=$k$ then
                        Let Justifyvalue($j$)$\leftarrow$($v_1$,$v_2$)
                    else
                        Let JustifyValue($j$)$\leftarrow$($v_1$,'X')
                        Let JustifyValue($k$)$\leftarrow$('X',$v_2$)
                    Add $j$ to EventList[level($j$)]
                    Add $k$ to EventList[level($k$)]
                else if $v_1$ is a controlling value of $g$ then
                    Find an input, $j$, of $g$ that satisfy $v_1$
                    Let JustifyValue($j$)$\leftarrow$($v_1$,$v_2$)
                    Add $j$ to EventList[level($j$)]
                    for every input $k$ of $g$ such that $k \neq j$ do
                        Let JustifyValue($k$)$\leftarrow$('X',$v_2$)
                        Add $k$ to EventList[level($k$)]
                else if $v_2$ is a controlling value of $g$ then
                    Find an input, $j$, of $g$ that satisfy $v_2$
                    Let JustifyValue($j$)$\leftarrow$($v_1$,$v_2$)
                    Add $j$ to EventList[level($j$)]
                    for every input $k$ of $g$ such that $k \neq j$ do
                        Let JustifyValue($k$)$\leftarrow$($v_1$,'X')
                        Add $k$ to EventList[level($k$)]
                else
                    for every input, $j$, of $g$ do
                        Let JustifyValue($j$)$\leftarrow$($v_1$,$v_2$)
                        Add $i$ to EventList[level($j$)]
```

list. If v_1/v_2 can't be satisfied by a single input, then it will be justified through two different inputs. In case only v_1 is a controlling value of g, the algorithm will find an input j with a fault-free value that satisfies v_1. Since v_2 is a non-controlling value (or an 'X'), then all inputs of g are required to justify this value. Therefore, input j is added to the event list to justify the value v_1/v_2, while other inputs are added to the event list to justify the value X/v_2. In the third case, only v_2 is controlling value of g. This can be handled exactly as done in the previous case except that v_2 is justified through one input, while v_1 is justified through all the inputs of g. Finally, if neither v_1 nor v_2 is a controlling value of g, then all the inputs of g are required to justify the value v_1/v_2. Hence, all inputs of g are added to the event list.

5 Selection Criteria

When justifying a controlling value through the inputs of a given gate, there could be more than one choice. In this case the priority is given to the input that is already selected to justify other gates. Otherwise, cost functions are used to guide the selection. The cost functions give a relative measure on the number of primary inputs required to justify a given value. Hence, they can guide the relaxation procedure to justify the required values with the smallest number of assignments on the primary inputs.

The cost functions proposed in [9] combine the *regular* recursive controllability cost functions [10] with new cost functions called *fanout-based* cost functions. The regular cost functions are computed as follows. For every gate g, we compute two cost functions $C_{reg0}(g)$ and $C_{reg1}(g)$. For example, if g is an AND gate with i inputs, then the cost functions are computed as:

$$C_{reg0}(g) = \min_i C_{reg0}(i)$$
$$C_{reg1}(g) = \sum_i C_{reg1}(i)$$

These costs functions are computed for other gates in a similar manner. The fanout-based cost functions can be computed for an AND gate as follows. Let g be an AND gate with i inputs. Let $F(g)$ denotes the number of fanout branches of g. Then, the fanout-based cost functions are computed as:

$$C_{fan0}(g) = \frac{\min_i C_{fan0}(i)}{F(g)}$$
$$C_{fan1}(g) = \frac{\sum_i C_{fan1}(i)}{F(g)}$$

The 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

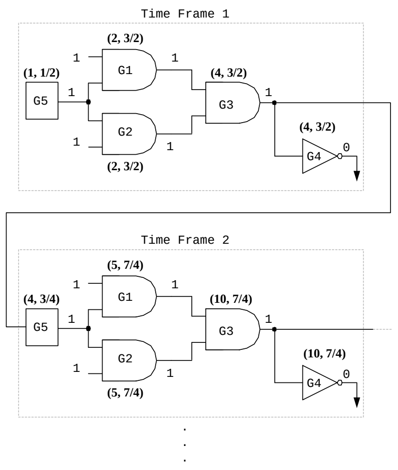

Figure 4: Circuit of Example 4.

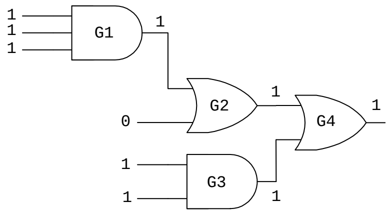

Figure 5: Circuit of Example 5.

several required values. In general, the fanout-based cost functions provide better selection criterion than the regular fanout cost functions. However, there are some cases where the regular fanout cost functions can perform better than the fanout-based cost functions [9]. To take advantage of both cost functions, a weighted sum cost function of the two cost functions was proposed in [9]. The combined cost functions are defined as follows:

$$C_0(g) = A \cdot C_{reg0}(g) + B \cdot C_{fan0}(g)$$
$$C_1(g) = A \cdot C_{reg1}(g) + B \cdot C_{fan1}(g)$$

In synchronous sequential circuits, the controllability values of the circuit in one time frame depend on the controllability values computed in the current frame as well as the values computed in the previous frames. Therefore, the controllability values should be computed in an iterative manner starting from the first time frame. However, the iterative computation of the controllability values may cause the regular cost to grow much faster than the fanout-based cost such that the effect of the second cost in the weighted sum becomes negligible. This is illustrated in the following example.

Example 4: Consider the iterative model shown in Figure 4. The controllability values of each gate are shown as a tuple of two values. The first value represents the regular cost, while the second value represents the fanout-based cost. Let the regular and fanout-based costs of all primary inputs equal to 1. Assume that the regular and fanout-based costs of the memory-element in the first time frame equal to 1 and 0.5 respectively. Then, in the first time frame, the regular and fanout-based costs of ($G3 = 1$) are 4 and 1.5 respectively. After 10 time frames, the regular cost of ($G3 = 1$) becomes 3070, while the fanout-based cost becomes $\frac{2047}{1024} \approx 2$.

The huge difference between the two costs in the previous example is due to the reconverging fanout branches of $G5$. Therefore, the regular cost of a memory-element with reconverging fanout branches should be adjusted to reduce the difference between the two costs. This can be done as follows. Let g be a memeory-element with n fanout branches. Assume that m out of the n fanout branches reconverge at some gate in the circuit, then the regular cost of every one of these branches equals to the regular cost of g divided by m. In Figure 4, both branches of the flip-flop $G5$ reconverge at the gate $G3$. Therefore, the regular cost of each branch is computed as the regular cost of the memory-element divided by 2. After adjusting the regular costs on the fanout branches of $G5$, the regular cost of ($G3 = 1$) becomes 3 in the first time frame and 21 in the 10th time frame.

The cost functions described so far compute the controllability values of a gate assuming general values on the gate inputs. Controllability values computed based on this assumption are less accurate than those computed based on the actual logical values. This is illustrated in the following example.

Example 5: Consider the circuit shown in Figure 5. If we compute cost of 1 (C_1) for each gate assuming general values on the input lines, then we get the following values: $C_1(G1) = 3$, $C_1(G2) = 1$, $C_1(G3) = 2$, and $C_1(G4) = 1$. These values suggest to justify the assignment $G4 = 1$ through $G2$ which results in three assignments on the primary inputs. Now, if we compute the controllability values based on the actual logical values, then we get the following values: $C_1(G1) = 3$, $C_1(G2) = 3$, $C_1(G3) = 2$, and $C_1(G4) = 2$. In this case, $G3 = 1$ will be selected to justify the assignment $G4 = 1$. This assignment requires only two assignments on the primary inputs.

In our work, cost functions are computed based on the actual values.

6 Experimental Results

In order to demonstrate the effectiveness of our proposed test relaxation technique, we have performed some experiments on a number of the ISCAS89 benchmark cir-

Table 1: Test relaxation comparison between the proposed technique and the bitwise-relaxation method.

	Percentage of X's			**CPU Time (seconds)**	
Circuit	**Bitwise-Relaxation**	**Proposed Technique**	**Diff.**	**Bitwise-Relaxation**	**Proposed Technique**
s1423	69.922/74.392	63.020	6.902/11.37	943	1.750
s1488	76.154/81.090	72.244	3.910/8.846	12553	2.417
s1494	76.295/82.962	72.741	3.554/10.22	13146	3.100
s3271	83.894/85.527	81.908	1.986/3.619	87726	8.033
s3330	87.738/90.082	85.506	2.232/4.576	115585	5.633
s3384	78.579/81.655	77.755	0.824/3.900	16549	2.533
s4863	84.832/87.542	81.735	3.097/5.807	162894	7.800
s5378	87.738/88.969	86.056	1.682/2.913	218137	20.35
AVG.	80.644/84.027	77.621	3.023/6.406		

cuits. The experiments were run on a SUN Ultra60 (Ultra-Sparc II 450MHz) with a RAM of 512MB. We have used test sets generated by HITEC[11]. In addition to that, we have used the fault simulator HOPE[12] for fault simulation purposes.

In Table 1, we compare the proposed test relaxation technique with the bitwise-relaxation method. The two techniques are compared in terms of the percentage of X's extracted, and the CPU time taken for relaxation. It is important to point out here that in order to have a fair comparison between our technique and the bitwise-relaxation method, we have constrained the bitwise-relaxation method such that all faults detected at a particular time frame remain detected in the same time frame after relaxation. However, the results obtained by both constrained and unconstrained bitwise-relaxation are shown in Table 1.

It is clear that, for all the circuits, the CPU time taken by our technique is less than that of the bitwise-relaxation method by several orders of magnitude. The bitwise-relaxation method requires enormous CPU times, and hence is impractical for large circuits.

The percentage of X's obtained by our technique is also close to the percentage of X's obtained by the bitwise-relaxation method for most of the circuits. The difference in the percentage of X's ranges between 1% and 7% (3% and 11% when compared with the unconstrained bitwise-relaxation method), while the average difference is about 3% (6% when compared with the unconstrained bitwise-relaxation method). It should be observed that the bitwise-relaxation method implicitly chooses the output for detecting a fault that maximizes the number of X's according to the order used. However, our technique does not do any optimization in selecting the best output for detecting a fault. This can be investigated in future work.

Table 2 shows the effect of varying the weights of the regular and fanout-based cost functions on the percentage of X's. Note that weight A is for the adjusted regular cost function and weight B is for the fanout-based cost function. As can be seen from the table, the use of cost functions results in higher percentage of X's. Also, it is worth mentioning here that neither the adjusted regular cost function nor the fanout-based cost function consistently performs better for all the circuits. However, when both cost functions are combined, better results are obtained. The table, also, shows that a weight of 1 for the adjusted regular cost function and a weight of 90 for the fanout-based cost function seems to be a good heuristic as it gives the highest percentage of X's on average.

Table3 shows the percentage of X's obtained using unadjusted cost functions with different weights. The results obtained for most of the circuits are close to those in Table 2 except for the circuits **s1488** and **s1494**. These two circuits show inconsistent results as compared to the other circuits. To see this clearly, let's consider the percentage of X's obtained using the weights $\{A = 0, B = 1\}$ and $\{A = 1, B = 50\}$. While the weights $\{A = 1, B = 50\}$ result in an enormous drop in the percentage of X's for these two circuits, they improve the results obtained for the remaining circuits. This inconsistency occurs because the regular cost function in these two circuits grows much faster than the fanout-based cost function. This problem can be avoided by adjusting the regular cost function to account for reconverging fanouts in memory-elements as explained in Example 4.

7 Conclusion

In this paper, we have proposed an efficient test relaxation technique for synchronous sequential circuits. Comparison between our technique and the bitwise-relaxation method for a number of ISCAS89 benchmarks showed that our technique is faster by several orders of magnitude. The percentage of X's obtained by our technique is close to the percentage of X's obtained by the bitwise-relaxation method. The difference is about 3% on average. Having a relaxed test set increases the effectiveness of both compression and compaction techniques. Also, the proposed

Table 2: Cost function effect on the extracted percentage of X's.

Circuit	**A=0 B=0**	**A=0 B=1**	**A=1 B=0**	**A=1 B=10**	**A=1 B=30**	**A=1 B=50**	**A=1 B=70**	**A=1 B=90**
s1423	37.882	50.863	57.059	62.431	63.686	63.961	**64.093**	63.020
s1488	44.448	**72.457**	56.624	66.218	69.968	71.767	71.571	72.244
s1494	43.515	72.661	57.410	66.687	70.502	71.767	72.098	**72.741**
s3271	57.361	78.860	**82.060**	82.017	82.033	81.979	81.892	81.908
s3330	66.548	85.251	84.805	85.446	85.407	85.484	**85.506**	**85.506**
s3384	69.247	71.703	77.755	**77.799**	77.784	77.755	77.755	77.755
s4863	72.114	78.934	**83.406**	82.846	82.582	82.393	82.038	81.735
s5378	77.788	85.692	82.130	84.110	85.053	85.085	85.094	**86.056**
Avg.	58.613	74.553	72.656	75.944	77.127	77.459	77.499	**77.621**

Table 3: Percentage of X's obtained using different weights of the unadjusted cost functions.

Circuit	**A=0 B=0**	**A=0 B=1**	**A=1 B=0**	**A=1 B=10**	**A=1 B=30**	**A=1 B=50**	**A=1 B=70**	**A=1 B=90**
s1423	37.882	50.863	60.314	64.157	66.000	66.784	66.902	66.980
s1488	43.515	**72.521**	45.288	47.714	48.152	48.942	48.622	48.248
s1494	44.448	**72.671**	47.500	50.050	50.512	51.396	51.084	50.552
s3271	57.361	81.062	82.060	82.315	82.445	82.478	82.494	82.462
s3330	66.548	85.251	85.182	85.169	85.342	85.476	85.536	85.584
s3384	69.247	71.790	77.755	77.799	77.784	77.755	77.755	77.755
s4863	72.114	77.630	83.406	83.287	83.173	83.169	83.126	83.094
s5378	77.788	85.692	84.771	86.075	86.350	86.347	86.269	86.241
AVG.	58.613	74.685	70.785	72.071	72.470	72.793	72.724	72.615

technique can be used for extracting self-synchronizing test sequences. This will be investigated in future work.

Acknowledgment

The authors would like to thank King Fahd University of Petroleum and Minerals for support.

References

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

[2] S. Hellebrand, S. Tarnick, J. Rajski, and B. Courtois, "Generation of Vector Patterns Through Reseeding of Multiple-Polynomial Feedback Shift Registers", in *IEEE International Test Conference*, Sep. 1992, pp. 120–129.

[3] A. Jas and N. Touba, "Test Vector Decompression via Cyclical Scan Chains and Its Application to Testing Core-Based Designs", in *Proc. International Test Conference*, 1998, pp. 458–464.

[4] A. Chandra and K. Chakrabarty, "Test Data Compression for System-On-a-Chip using Golomb Codes", in *Proc. of IEEE VLSI Test Symposium*, 2000, pp. 113–120.

[5] A. Chandra and K. Chakrabarty, "Frequency-directed run-length (FDR) codes with application to system-on-a-chip test data compression ", in *19th IEEE Proceedings on. VTS*, 2001, pp. 42–47.

[6] T. Yamaguchi, M. Tilgner, M. Ishida and D. S. Ha, "An Efficient Method for Compressing Test Data", in *Proc. International Test Conference*, Nov. 1997, pp. 79–88.

[7] R. Roy, T. Niermann, J. Patel, J. Abraham, and R. Saleh, "Compaction of ATPG-Generated Test Sequences for Sequential Circuits", Nov. 1988, pp. 382–385.

[8] S. Kajihara and K. Miyase, "On Identifying Don't Care Inputs of Test Patterns for Combinational Circuits", in *Proc. IEEE ICCAD*, Nov. 2001, pp. 364–369.

[9] A. El-Maleh and A. Al-Suwaiyan, "An Efficient Test Relaxation Technique for Combinational & Full-Scan Sequential Circuits", in *Proc. IEEE VLSI Test Symposium*, 2002, pp. 53–59.

[10] M. Abramovici, M. Breuer and A. Friedman, *Digital System Testing and Testable Design*, IEEE Press, 1990.

[11] Thomas M. Niermann and Janak H. Patel, "HITEC: A test generation package for sequential circuits", in *Proc. of the European Conference on Design Automation (EDAC)*, 1991, pp. 214–218.

[12] H. K. Lee and D. S. Ha, "HOPE: An Effecient Parallel Fault Simulator for Synchronous Sequential Circuits", *IEEE Trans. on Computer Aided Design*, vol. 15, no. 9, pp. 1048–1058, Sep. 1996.