An Efficient Test Relaxation Technique for Synchronous Sequential Circuits

Aiman El-Maleh and Khaled Al-Utaibi

***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. Test-set *relaxation* 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.**

I. 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. The objective of test set compression is to reduce the number of bits needed to represent the test set. Several test compression techniques have been proposed [1]–[15]. In test compaction, the number of test vectors is reduced into a smaller number that achieves the same fault coverage. Test compaction techniques can be classified into two categories: *dynamic compaction* and *static compaction*. Dynamic compaction schemes such as [16]–[19] try to reduce the number of test vectors during test vector generation. Static compaction schemes, on the other hand, perform compaction on test sequences after they are generated. Several static test compaction techniques have been proposed for synchronous sequential circuits. The techniques proposed in [20] use overlapping of self-initializing test sequences. Four compaction techniques based on insertion, omission, selection and restoration have been proposed in [21], [22]. The technique in [23] compacts test sequences by removing inert subsequences under certain conditions.

Manuscript received December 20, 2003; revised December 30, 2003. This work was supported by King Fahd University of Petroleum and Minerals, Dharan 31261, Saudi Arabia.

A. El-Maleh and K. Al-Utaibi are with King Fahd University of Petroleum and Minerals. Emails:{aimane, alutaibi}@ccse.kfupm.edu.sa

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 turn 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 [20], 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 efficiently identify self-initializing test sequences, which have interesting applications in test sequence compaction for sequential circuits. A test sequence is said to be self-initializing if the values of the memory-elements in the first time frame of the sequence are all X's. In other words, the values of the memory-elements of the first time frame do not affect the faults detected by the sequence. Thus, self-initializing test sequences can be identified by finding time frames where the values of all memory-elements are not required for fault detection. As will be shown in this paper, the proposed test relaxation technique can identify circuit lines that have no effect on the detected faults.

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 an example. 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.

II. Problem Definition and Illustrative Example

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 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 ATPG. It is known that any ATPG in generating a test for a fault generates test cubes including the required assignments to excite and propagate the fault. These test cubes are filled with random values to allow the detection of other faults. If the random filling is disabled, a relaxed test can be obtained. Using this approach to generate a relaxed test set has several disadvantages. First, to get a relaxed test it is not possible to take advantages of random test pattern generation. It is well-known that random test pattern generation is an integrated phase of any ATPG as it can detect a large percentage of the faults (typically between 50 and 80 percent [24]) in combinational circuits in a much faster time than deterministic test pattern generation. Disabling random test pattern generation and random filling will certainly slow down the ATPG. For sequential circuits, due to the complexity of the problem, many heuristics are used to obtain as high fault coverage as possible. For example, genetic algorithm which is based on random test generation has been used in sequential ATPG. Such techniques will not generate a relaxed test set.

The advantage of the proposed technique is that it is ATPG independent. The ATPG can freely use all heuristics to achieve the highest fault coverage in the least time. The generated test can also be compacted. Then, test relaxation is applied to relax the test set.

Recently, two test relaxation techniques for combinational and full-scan sequential circuits were proposed in [25], [26]. 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.

The techniques proposed in [25], [26] justify detected faults based on fault-free 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.

Our proposed technique is based on fault-free/faulty value justification. 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-free/faulty values necessary to detect 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.

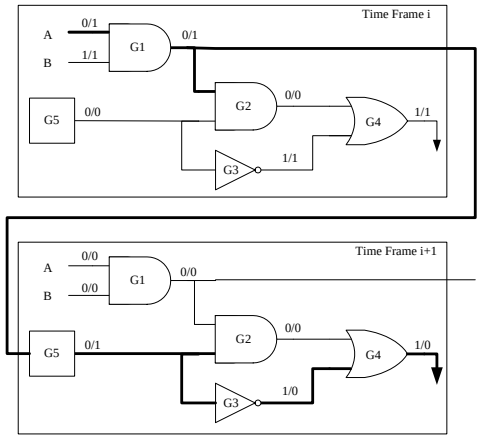

Fig. 1. Circuit of Example 1.

Example 1: Consider the iterative-array-model shown in Fig. 1. 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 $B = X/1$. Since the fault-free/faulty values of both inputs are required, none of these inputs can be relaxed

in this time frame.

III. 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 frames 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. Seven lists are used to store the required information: *POJustificationList* $[1 \cdots n]$, *PPOJustificationList* $[1 \cdots n]$, *MEJustificationList* $[1 \cdots F]$, *FaultPropagationList* $[1 \cdots F]$, *PPInputList* $[1 \cdots n]$ $[1 \cdots D]$, *EventList* $[1 \cdots L]$, and *RelaxedTestSet* $[1 \cdots n]$ $[1 \cdots NPI]$, where n is the number of test vectors to be relaxed, F is the total number of faults in the given circuit, L is the number of levels in the given circuit, NPI is the number of primary inputs of the given circuit, and D is the number of memory-elements in the given circuit. The purpose of each one of these seven 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, their faulty values, and the corresponding time frame are added to the *FaultPropagationList*. 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, the fault is added to the *PPOJustificationList*$[i-1]$ and the memory-elements along with their required fault-free/faulty values are added to *MEJustificationList*$[f]$, and the justification of f will continue during time frame $i-1$. The purpose of the *PPInputList* is to store logical values of the memory elements in every time frame. This enables us to perform logic simulation of any time frame independent of other time frames. 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. 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.

Fig. 2 shows an outline of the proposed test relaxation technique which consists of three phases. The first phase initializes the seven lists.

Fault simulation is performed in the second phase starting from the first time frame up to the last time frame n. The initial states of all memory elements are assumed to be X's. The purpose of this process is 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 and the corresponding time frame are added to *FaultPropagationList*$[f]$. The information in this list will be used to compute faulty values of the circuit during the justification phase. Another important operation that is performed in this phase is to store logical values of the memory-elements into the *PPInputList* for all time frames.

The third phase starts from the last time frame (n) 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 $i+1$ and tries to justify it in the current time frame (i.e. time frame i). 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 below.

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 $i-1$. These values are stored in the *FaultPropagationList* in the second phase. 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.

Fig. 3 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 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 required values 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 (e.g. time frame i). Therefore, the memory-element together with its *JustifyValue* are added to *MEJustificationList*$[f]$, and the fault f is added to the *PPOJustificationList*$[i-1]$.

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

Algorithm *Main*()

```
/* initialization phase */
for every level, l, of the given circuit do
    EventList[l] ← φ
for every fault, f, in the fault list of the given circuit do
    FaultPropagationList[f] ← φ
    MEJustificationList[f] ← φ
for every test vector i do
    POJustificationList[i] ← φ
    PPOJustificationList[i] ← φ
    for every primary input j do
        RelaxedTestSet[i][j] ← 'X'
    for every memory-element j do
        PPInputList[i][j] ← 'X'

/* Fault simulation phase*/
for i ← 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 f
           together with their faulty values and the
           corresponding time frame i to FaultPropa-
           gationList[f] */
    /* store logical values of all memory-elements into
       PPInputList[i] */

/* Fault justification phase */
for i ← n downto 1 do
    logic simulate the circuit under the test vector i
    whilePPOJustificationList[i] ≠ φ do
        /* remove f from PPOJustificationList[i] */
        /* compute faulty values of the circuit based on
           the injected faultf and the fault propagation
           stored in FaultPropagationList[f] */
        while MEJustificationList[f]≠ φ do
            /* remove memory-element d from MEJusti-
               ficationList[f] */
            /* let j be the input of d & add j to Event-
               List[level(j)] */
        Justify(f, i)
        whilePOJustificationList[i] ≠ φ do
            /* remove f from POJustificationList[i] */
            /* compute faulty values of the circuit based on
               the injected fault f and the fault propagation
               stored in FaultPropagationList[f] */
            /* let j be a primary-output at which the fault f
               gets detected & add j to EventList[level(j)] */
            Justify(f, i)
```

Fig. 2. Main algorithm.

Algorithm *Justify*(f,i)

```
for every level, l, of the given circuit do
    while EventList[l]≠ φ do
    /* remove gate g from the EventList[l] */
    (v1,v2)←JustifyValue(g)
    if g is (NOT|NAND|NOR) then
        (v1,v2) ← (v̄1,v̄2)
    case g is
        (1) PI:
            if v1 ≠'X' then
                RelaxedTestSet[i][g]← v1
            else if v2 ≠'X' then
                RelaxedTestSet[i][g]← v2
        (2) DFF:
            /* add f to PPOJustificationList[i − 1] */
            /* add g and JustifyValue(g) to MEJustifica-
               tionList[f] */
        (3) BUF|NOT:
            /* let j be the input of g */
            JustifyValue(j)←(v1, v2)
            /* add j to EventList[level(j)] */
        (4) XOR|XNOR:
            for every input, j, of g do
                (v1, v2) ← (GoodValue(j), FaultyValue(j))
                JustifyValue(j) ← (v1, v2)
                /* add j to EventList[level(j)] */
        (5) AND|OR|NAND|NOR:
            if v1 and v2 are controlling values of g then
                /* find an input, j, of g that satisfies v1 */
                /* find an input, k, of g that satisfies v2 */
                if j=k then
                    JustifyValue(j)←(v1,v2)
                else
                    JustifyValue(j)←(v1,'X')
                    JustifyValue(k)←('X',v2)
                /* add j to EventList[level(j)] */
                /* add k to EventList[level(k)] */
            else if v1 is a controlling value of g then
                /* find an input, j, of g that satisfy v1 */
                JustifyValue(j)←(v1,v2)
                /* add j to EventList[level(j)] */
                for every input k of g such that k ≠ j do
                    JustifyValue(k)←('X',v2)
                    /* add k to EventList[level(k)] */
            else if v2 is a controlling value of g then
                /* find an input, j, of g that satisfy v2 */
                JustifyValue(j)←(v1,v2)
                /* add j to EventList[level(j)] */
                for every input k of g such that k ≠ j do
                    JustifyValue(k)←(v1,'X')
                    /* add k to EventList[level(k)] */
            else
                for every input, j, of g do
                    JustifyValue(j)←(v1,v2)
                    /* add j to EventList[level(j)] */
```

Fig. 3. Justify algorithm.

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 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.

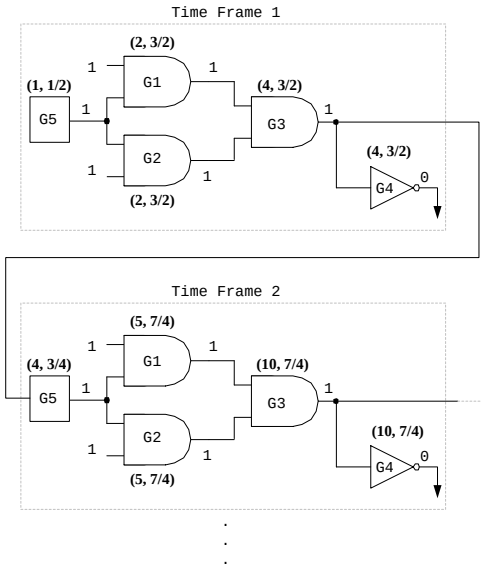

Fig. 4. Circuit of Example 2.

IV. 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. 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 [26] combine the *regular* recursive controllability cost functions [24] 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 $\{I_1, I_2, \cdots, I_i\}$, then the cost functions are computed as:

$$C_{reg0}(g) = \min_i C_{reg0}(I_i) \tag{1}$$

$$C_{reg1}(g) = \sum_i C_{reg1}(I_i) \tag{2}$$

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 $\{I_1, I_2, \cdots, I_i\}$. Let $F(g)$ denote the number of fanout branches of g. Then, the fanout-based cost functions are computed as:

$$C_{fan0}(g) = \frac{\min_i C_{fan0}(I_i)}{F(g)} \tag{3}$$

$$C_{fan1}(g) = \frac{\sum_i C_{fan1}(I_i)}{F(g)} \tag{4}$$

It is important to point out here that the cost of a primary-input is assumed to be 1 in the regular cost function and $1/F(g)$ in the fanout-based cost function. 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 several required values. In general, the fanout-based cost functions provide better selection criterion than the regular cost functions. However, there are some cases where the regular cost functions can perform better than the fanout-based cost functions [26]. To take advantage of both cost functions, a weighted sum cost function of the two cost functions was proposed in [26]. The combined cost functions are defined as shown below, where A is the weight of the regular cost function and B is the weight of the fanout-based cost function.

$$C_0(g) = A \cdot C_{reg0}(g) + B \cdot C_{fan0}(g) \tag{5}$$

$$C_1(g) = A \cdot C_{reg1}(g) + B \cdot C_{fan1}(g) \tag{6}$$

In synchronous sequential circuits, the controllability values of the circuit in one time frame depend on the controllability values computed in the current time frame as well as the values computed in the previous time frames. Therefore, the controllability values should be computed in an iterative manner starting from the first time frame up to the last 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 2: Consider the iterative model shown in Fig. 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 previ-

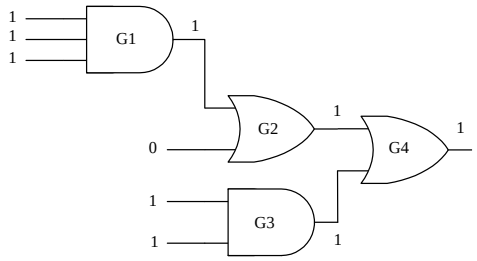

Fig. 5. Circuit of Example 3.

ous 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. The cost of the other $n-m$ non-reconverging branches will be the regular cost of g. In Fig. 2, 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, i.e., the cost of 1 and the cost of 0 on all inputs is assumed to be 1. Controllability values computed based on this assumption are less accurate than those computed based on the actual logical values as illustrated in Example 3. Note that if a primary input has a logic value 1(0), then the cost of 0(1) for this input based on the actual logical values is ∞.

Example 3: Consider the circuit shown in Fig. 5. If we compute the 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.

V. 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 circuits shown in Table I. The first column gives the name of the benchmark circuit. Columns 2 to 8 give the number of primary inputs, the number of primary outputs, the number of D flip-flops, the total number of gates, the number of applied test vectors, the number of collapsed faults (CFs), and the number of detected faults(DFs), respectively. It is important to point out here that the last three circuits are partially scanned using OPUS [27] to increase their testability. Up to 30% of the flip-flops are scaned based on SCOAP testability measures after breaking all the loops in the circuit. The experiments were run on a SUN Ultra60 (UltraSparc II 450MHz) with a RAM of 512MB. We have used test sets generated by HITEC [28]. In addition to that, we have used the fault simulator HOPE [29] for fault simulation purposes.

TABLE I
Benchmark circuits.

Circuit Name	No. I/Ps	No. O/Ps	No. FFs	No. Gates	No. TVs	No. CFs	No. DFs
s1423	17	5	7	490	150	1515	723
s1488	8	19	6	550	1245	1506	1453
s1494	8	19	6	558	1170	1486	1444
s3271	26	14	116	1035	709	3270	3238
s3330	40	73	132	815	578	2870	2103
s3384	43	26	183	1070	161	3380	2996
s4863	49	16	104	1600	518	4764	4633
s5378	35	49	179	1004	912	3231	1372
s15850.1s	237	310	374	9772	8478	11725	16618
s38417s	518	596	1146	22179	7526	31180	25282
s38584s	361	627	1103	19253	11353	36303	28348

In Table II, 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 II.

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. In addition to that, the unconstrained bitwise-relaxation

TABLE II

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
s15850.1s	-	90.195	-	-	513.7
s38417s	-	93.988	-	-	1648
s38584s	-	92.272	-	-	1764

method relaxes the test sequence in such a way that each fault is detected by the last possible detecting test sequence. This increases the number of X's extracted as easy to detect faults are detected by test sequences generated for hard-to-detect faults.

Table III 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.

Table IV 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 III 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 2.

Table V shows the percentage of X's obtained using cost functions based on general values. If we compare the results in this table with those in Table III, we find that cost functions based on actual values extract more X's for most of the circuits, especially for the circuits **s1423** and **s4863**. Using cost functions based on actual values achieves an improvement of 15.26% for **s1423** and 7.706% for **s4863** under the weights $\{A = 1,\ B = 90\}$. The average difference between the percentage of X's obtained using the actual values and those obtained using the general values is more than 5%.

As shown in [30], the time complexity of our proposed test relaxation technique is $O(n \times F \times G)$; where n is the number of test vectors, F is the number of faults, and G is the number of gates in the given circuit. This is the same as the time complexity of fault simulation. Also, the space complexity is $O(n \times D \times F)$; where D is the number of memory-elements in the given circuit. Note that this worst case complexity occurs when all faults are excited in the first time frame, propagated to every time frame through all memory-elements, and not detected until the last time frame. However, in practice, a fault propagates through a portion of the time frames and through a fraction of the memory-elements.

Table VI compares the space complexity and the actual memory usage of the proposed technique for the considered circuits. As can be seen from this table, the memory usage by our test relaxation technique is significantly less than the worst case requirement. It should be observed that the memory usage by our technique can be reduced by storing only information about propagated faults from the time of their excitation until their detection. Currently, our technique stores all faults that get excited and propagated even if they are not detected or propagated to a primary output. Furthermore, the memory requirement can be reduced significantly by partitioning the fault list and performing test relaxation for each partition separately. The relaxed test is then obtained by the intersection of the relaxed test of each partition. Note that test relaxation for different fault partitions can be done in parallel, hence, speeding up the overall test time. These ideas will be investigated in future work.

VI. 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

TABLE III

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	**64.039**	63.020
s1488	43.515	**72.457**	56.624	66.218	69.968	71.250	71.571	72.244
s1494	44.448	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**
s15850.1s	80.982	87.364	86.274	88.594	88.880	89.198	**90.888**	90.195
s38417s	85.958	90.990	87.112	91.924	92.226	93.353	**93.988**	**93.988**
s38584s	83.920	91.801	87.455	90.057	91.152	91.441	92.071	**92.272**
AVG	65.433	78.780	76.554	79.830	80.843	81.242	81.540	**81.584**

TABLE IV

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
s15850.1s	80.982	87.423	86.711	89.131	89.659	90.077	92.032	91.547
s38417s	85.958	91.002	87.123	91.938	92.783	93.895	94.863	94.167
s38584s	83.920	91.801	87.486	90.074	91.185	91.462	92.174	92.416
AVG	65.433	78.882	75.236	77.064	77.580	77.980	78.259	78.095

TABLE V

PERCENTAGE OF X'S OBTAINED USING DIFFERENT WEIGHTS OF GENERAL-VALUES COST FUNCTIONS.

CKT NAME	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	47.176	**45.569**	**49.569**	**48.863**	**48.745**	**47.765**	**47.765**
s1488	43.515	65.556	70.150	68.365	68.301	68.301	68.226	68.226
s1494	44.448	66.888	72.339	70.633	70.592	70.592	70.552	70.552
s3271	57.361	66.150	82.174	80.975	78.024	77.748	77.362	76.283
s3330	66.548	81.315	84.619	84.931	83.382	83.110	82.375	82.535
s3384	69.247	71.414	77.842	77.784	74.014	73.393	73.393	72.916
s4863	72.114	74.127	**83.102**	**79.009**	**76.759**	**74.576**	**74.163**	**74.029**
s5378	77.788	85.423	82.303	84.207	85.069	86.012	86.012	86.012
s15850.1s	80.982	87.801	86.391	88.696	88.877	89.181	90.842	90.136
s38417s	85.958	87.771	86.639	87.039	86.243	86.338	86.257	86.353
s38584s	83.920	87.826	87.080	87.883	87.368	87.416	87.699	87.613
AVG	65.433	74.677	78.019	78.099	77.045	76.856	76.786	76.584

TABLE VI
MEMORY USAGE OF THE PROPOSED TECHNIQUE COMPARED TO THE SPACE COMPLEXITY.

Circuit Name	No TVs n	No. FFs D	No. Faults F	Space Complexity $n \times D \times F$	Memory Usage (Bytes)
s1423	150	7	1515	1590750	3499008
s1488	1245	6	1506	11249820	1067008
s1494	1170	6	1486	10431720	1217536
s3271	709	116	3270	268937880	6712320
s3330	578	132	2870	218969520	4657152
s3384	161	183	3380	99584940	4455424
s4863	518	104	4764	256646208	4111360
s5378	912	179	3231	527454288	20268032
s15850.1s	8478	374	11725	37177301700	73940992
s38417s	7526	1146	31180	2.68921E+11	253977600
s38584s	11353	1103	36303	4.54599E+11	362776576

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. We have demonstrated that the use of cost functions has a significant impact on the percentage of X's extracted.

Having a relaxed test set increases the effectiveness of both compression and compaction techniques. Also, the proposed technique can be used for extracting self-synchronizing test sequences which has interesting applications in test sequence compaction for sequential circuits, which will be investigated in future work.

ACKNOWLEDGMENT

The authors would like to thank King Fhad 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] ——, "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] V. Iyengar, K. Chakrabarty and B. Murray, "Huffman Encoding of Test Sets for Sequential Circuits," *IEEE Trans. on Instrumentation and Measurement*, vol. 47, no. 1, pp. 21–25, Feb. 1998.

[8] A. El-Maleh, S. Zahir, and E. Khan, "A Geometric-Primitive-Based Compression Scheme for Testing Systems-on-a-Chip," in *Proc. IEEE VLSI Test Symposium*, Apr. 2001, pp. 54–61.

[9] A. El-Maleh and R. Al-Abaji, "Extended Frequency-Directed Run Length Code with Improved Application to System-on-a-chip Test Data Compression," in *Proc. of the 9th IEEE International Conference on Electronics, Circuits and Systems*, Sep. 2002, pp. 449–452.

[10] A. Jas, J. G. Dastidar and N. Touba, "Scan Vector Compression/Decompression Using Statistical Coding," in *Proc. IEEE VLSI Test Symposium*, 1999, pp. 114–120.

[11] E. H. Volkerink, A. Khoche and S. Mira, "Packet-based Input Test Data Compression Techniques," in *Proc. International Test Conference*, 2002, pp. 154–163.

[12] A. R. Pandey, T. Patel, "Reconfiguration Technique for Reducing Test Time and Test Data Volume in Illinois Scan Architecture Based Designs," in *Proc. IEEE VLSI Test Symposium*, 2002, pp. 9–15.

[13] Lei Li and Krishnendu Chakrabarty, "Test Data Compression Using Dictionaries with Fixed-Length Indices," in *Proc. of IEEE VLSI Test Symposium*, 2003, pp. 219–224.

[14] I. Bayraktaroglu and A. Orailoglu, "Test volume and application time reduction through scan chain concealment," in *Proc. ACM/IEEE Design Automation Conf.*, 2001, pp. 151–155.

[15] P. T. Gonciari, B. Al-Hashimi and N. Nicolici, "Improving compression ratio, area overhead, and test application time for system-on-a-chip test data compression/decompression," in *Design Automation and Test in Europe Conf.*, 2002, pp. 604–611.

[16] I. Pomeranz and S. M. Reddy, "On Generating Compact Test Sequences for Synchronous Sequential Circuits," in *Proc. EURODAC*, Sep. 1995, pp. 105–110.

[17] S. Chakradhar and A. Raghunathan, "Bottleneck Removal Algorithm for Dynamic Compaction and Test Cycle Reduction," in *Proc. EURODAC*, Sep. 1995, pp. 98–104.

[18] I. Pomeranz and S. M. Reddy, "Dynamic Test Compaction for Synchronous Sequential Circuits using Static Compaction Techniques," in *Proc. 26th Fault-Tolerant Computing Symp.*, June 1996, pp. 53–61.

[19] E. M. Rudnick and J. H. Patel, "Simulation-based Techniques for Dynamic Test Sequence Compaction," in *Proc. Intl. Conf. on Computer Aided Design*, Nov. 1996, pp. 67–75.

[20] 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.

[21] I. Pomeranz and S. M. Reddy, "On static compaction of test sequences for synchronous sequential circuits," in *Proc. Design Automation Conf.*, 1996, pp. 215–220.

[22] ——, "Vector Restoration Based Static Compaction of Test Sequences for Synchronous Sequential Circuits," in *Proc. Int. Conf. on Computer Design*, Oct. 1997, pp. 360–365.

[23] M. S. Hsiao, E. M. Rudnick, and J. H. Patel, "Fast Static Compaction Algorithms for Sequential Circuit Test Vectors," *IEEE Trans. on Computers*, vol. 48, no. 3, pp. 311–322, March 1999.

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

[25] 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.

[26] 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.

[27] V. Chickermane and J. H. Patel, "A Fault Oriented Partial Scan Design Approach," in *Proc. of the Intl. Conf. on Computer-Aided Design*, Nov. 1991, pp. 400–403.

[28] 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.

[29] 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.

[30] K. Al-Utaibi, *An Efficient Test-Pattern Relaxation Technique for Synchronous Sequential Circuits*. M.S. thesis, King Fhad University of Petroleum and Minerals, Dhahran, 2002.