AN EFFICIENT TEST RELAXATION TECHNIQUE FOR COMBINATIONAL LOGIC CIRCUITS

Aiman El-Maleh [1], Ali Al-Suwaiyan [2]

1: Assistant Professor, Department of Computer Engineering, KFUPM, Dhahran, Saudi Arabia
2: Graduate Assistant, Department of Computer Engineering, KFUPM, Dhahran, Saudi Arabia

E-mail: aimane@ccse.kfupm.edu.sa

ABSTRACT

Reducing test data size is one of the major challenges in testing systems-on-a-chip. This can be achieved by test compaction and/or compression techniques. Having a partially specified or relaxed test set increases the effectiveness of compaction and compression techniques. In this paper, we propose a novel and efficient test relaxation technique for combinational circuits. It is based on critical path tracing and hence it may result in a reduction in the fault coverage. However, based on experimental results on ISCAS benchmark circuits, the drop in the fault coverage (if any) after relaxation is small for most of the circuits. The technique is faster than the brute-force test relaxation method by several orders of magnitude.

Keywords test compression, test compaction, system-on-a-chip, partially specified test set, fault simulators, critical path tracing.

الملخص

تصغير حجم بيانات الاختبارات للدوائر المتكاملة يعتبر من التحديات الأساسية في مجال اختبار الدوائر عالية التكامل. يمكن عمل ذلك بالضغط أو التقليص. وجود بيانات اختبار محددة جزئياً أو مرخاة يزيد من فعالية ضغط و تقليص بيانات الاختبار. نقدم في هذا البحث طريقة جديدة و فعالة لإرخاء بيانات الاختبار. هذه الطريقة مؤسسة على خوارزمية المسار الحرج وبالتالي فقد ينتج عنها نقص في تغطية الأعطال. لكن النقص على أية حال صغير لأغلب الدوائر التي أجريت عليها التجارب. الطريقة المقترحة تعطي نتائج مقاربة للطريقة التقليدية ولكنها أسرع منها و بفارق كبير.

1. INTRODUCTION

With today's VLSI technology, it is possible to build very large systems containing millions of transistors on a single chip. One of the challenges in testing a System-on-a-Chip (SOC) is dealing with the large volume of test data that must be stored in the tester memory, and transferred between the tester and the chip [Zorian et al.: 1998].

To reduce test storage requirements, test compaction and test compression can be used. 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. Examples of compaction algorithms can be found in [Hamzaoglu and Patel: 1998] [Chang and Lin: 1995] [Schulz et al.: 1988]. The objective of test set 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 [Chandra and Chakrabarty: 2000] [Chandra and Chakrabarty: 2001] [El-Maleh et al.: 2001] [Jas and Touba: 1998].

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. Furthermore, without the dynamic compaction option, ATPGs generally generate fully specified test sets. The problem of test set relaxation, i.e. extracting a partially specified test set from a fully specified one, has not been solved effectively in the literature. One obvious way to solve this problem is to use a brute-force technique, where we test for every bit of the test set whether changing it to an *x* reduces the fault coverage or not. This technique has a complexity of *O(nm)* fault simulation runs, where *n* is the width of one test vector, and *m* is the number of test vectors. Although only the newly detected faults by a vector are fault simulated, this technique is impractical for large circuits.

In this paper, we propose a novel and efficient test relaxation technique for combinational circuits. This technique is based on the critical path tracing (CRIPT) algorithm [Abramovici et al.: 1990]. A very important characteristic of this algorithm is that when a fault is detected by a given test vector, there exists (at least) one continuous critical path to a primary output. This property simplifies identifying the required values needed for propagating a fault effect to a primary output, and it is the main reason why we adopted CRIPT in our technique. As with CRIPT, the proposed technique is not exact in the sense that the fault coverage might be reduced after relaxation. However, as will be shown from experimental results, the drop in the fault coverage is small for most of the circuits. Compared to the brute-force method, our technique is faster by several orders of magnitude.

This paper is organized as follows. The next section illustrates our idea by an example. Section 3 formally describes our test relaxation algorithm. Experimental results are given in section 4. Finally, the paper ends by a conclusion.

2. AN ILLUSTRATIVE EXAMPLE

In this section, we demonstrate our proposed test relaxation technique by an example. Section 3 formally describes our algorithm. The following conventions are assumed.

Top

To indicate that a line l is stuck at value *v*, we use the notation *l/v*. When we say that line *l* is required, we mean that the value on line *l* is required.

Definition 1 *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* [Abramovici et al.: 1990].

Example 1: Consider the circuit shown in Figure 1. Suppose that we apply the test vector *ABCDE=00000*. Under this test, lines *G6, G5, G1, G4, G2, 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 *DE=00*. If we choose to satisfy *G3=0* by the assignment *C=0*, then *DE=00* is no longer necessary, and this implies that we can relax *CDE* to *0xx*. Similarly, if we choose to satisfy *G3=0* by the assignments *DE=00*, 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.

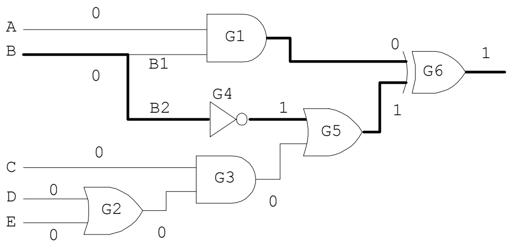

Figure 1: Circuit of Example 1.

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, we now formally describe our technique in the following section.

3. PROPOSED TECHNIQUE

Algorithm 1 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 2, checks for every stem *s* in the list, whether *s s*atisfies the condition or

Algorithm 1 Main 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 fault on s is newly detected then
9:            add it 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
```

Top

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 [Abramovici et al.: 1990], but it does one extra job, namely adding newly detected faults to the *NDF* list. The *Critical* function checks whether a stem is critical or not. This is done by fault simulation. After fault simulation is performed for stem *s*, all the primary outputs in which the fault on *s* propagates to, are saved in order to avoid repeating fault simulation under the same test vector. In other words, fault simulation is done only once for a given stem under a given test vector.

Once the *CS* and *NDF* lists are constructed, the algorithm marks reachable lines by the procedure *MarkReachableLines.* This is discussed in section 3.1. Then, the algorithm justifies the requirements by the procedure *MarkRequiredLines*, which is the topic of section 3.2. 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 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
```

3.1. 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. Let us have the following definitions.

Definition 2 *A line l is said to be reachable from a stem s if the fault effect in stem s reaches the line l.*

Definition 3 *A gate input is said to be sensitive in a test vector t if complementing its value changes the value of the gate output from v to* $\bar{v}$ *where* $v \in \{0,1\}$ [Abramovici et al.: 1990].

Algorithm 3 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 two lemmas provide the rules used by the function *Reachable(l,s)*.

Lemma 1 *Let l be the output of an AND, NAND, OR, or NOR gate. Then l is reachable from stem s iff one of the following conditions is satisfied:*

1. *Only sensitive inputs of l are reachable from stem s.*
2. *Only the non-sensitive inputs of l having controlling value are reachable from stem s, and none of the other gate inputs has an x value.*

Lemma 2 *Le l be the 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.*

Algorithm 3 MarkReachableLines()

```
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:           mark fanouts of l as reachable from s
10:          add fanouts of l to E
11:       end if
12:    end while
13: end for
```

3.2. Requirement Analysis

Algorithm 4 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 5. After this step is over, we will have a requirement list *L* to be justified.

Algorithm 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:     ForwardTrace(l)
5: end for
6: for every line l in L do
7:     justify(l)
8: end for
```

Algorithm 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
```

Algorithm 6 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 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 we justify only the reachable lines. 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 6 justify(l)

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

3.3. Selection Criteria

As has been illustrated in the previous sections, there could be several choices for justifying a required value. Our objective 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.

To achieve this objective, we use cost functions that provide a relative measure on the selection that reduces the number of required assignments on the PIs.

The well-known recursive controllability cost functions [Abramovici et al.: 1990] can be used for this purpose as they give a relative measure of the number of PI assignments required to justify a required value. These cost functions are accurate for fanout-free circuits; however, due to the existence of fanout, they 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 fanout-based cost functions. Note that these cost functions are different from the fanout-based cost functions given in [Abramovici et al.: 1990]. 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:)

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

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

These cost functions can be computed similarly for other gates.

Table 1: Benchmark circuits characteristics.

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

4. EXPERIMENTAL RESULTS

In order to demonstrate the effectiveness of our proposed test relaxation technique, we have performed experiments on a number of the largest ISCAS85 and full-scanned versions of ISCAS89 benchmark circuits shown in Table 1. In this table, the first column gives the names of the benchmark circuits. Columns 2 to 7 indicate the number of inputs, outputs, gates, levels, collapsed faults, 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 [Hamzaoglu and Patel: 1998]. The experiments were run on a SUN Ultra60 (UltraSparc II-450 MHZ) with a RAM of 512 MB.

In Table 2, we compare the proposed test set relaxation technique with the brute-force relaxation method. The first column in the table indicates 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. We have used the fault simulator HOPE [Lee and Ha: 1996] to

Top

determine the fault coverage of the used test sets. It is important to point out here that the fault coverage of the relaxed test set based on the brute-force method 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. However, the fault coverage of the relaxed test set based on our technique may be reduced. This is due to the approximate nature of the CRIPT algorithm on which our technique is based. The fault coverage of the relaxed test set based on our technique is equivalent to the fault coverage of the original test set as measured by CRIPT. As can be seen from the table, the drop in the fault coverage is small for most of the circuits.

In order to compensate for the drop in the fault coverage, the test vectors needed for detecting the undetected faults can be relaxed based on the brute-force method and then merged with the relaxed test set based on our technique.

It is very interesting to observe that the CPU time taken by our proposed technique is several orders of magnitude less than the brute-force method for most of the circuits. The brute-force method requires astronomical CPU times for large circuits and hence is impractical.

The percentage of *x's* obtained by our technique is also close to the percentage of *x's* obtained by the brute-force method for most of the circuits. The difference in the percentage of *x's* obtained ranges between 1% and 9%. For seven of the eight circuits, it is less than 4%. The advantage of using our proposed fanout-based cost functions is clearly illustrated in Table 2. For all the circuits, using the selection criteria increases the percentage of x's from 0.5% to 4%.

Table 2:Test relaxation comparison between the proposed technique and the brute-force method.

	Brute Force Test Relaxation			Proposed Test Relaxation					
				Selection Criteria			No Selection Criteria		
Circuit	FC Exact/CRIPT	%*x*	CPU (sec)	FC Exact/CRIPT	%*x*	CPU (sec)	FC Exact/CRIPT	%*x*	CPU (sec)
c5315	98.90/98.90	54.37	1192	98.90/98.90	51.99	3	98.90/98.90	48.89	2
c7552	98.26/98.12	55.45	6645	98.21/98.12	52.17	6	98.21/98.12	48.23	6
c2670	95.74/95.70	69.63	2757	95.74/95.70	68.36	1	95.74/95.70	66.03	1
s5378	99.13/99.02	74.14	7451	99.04/99.02	70.21	3	99.04/99.02	67.64	4
s9234.1	93.47/89.53	70.29	19837	90.00/89.53	68.57	5	90.07/89.53	66.69	5
s15850.1	96.68/96.46	80.96	87120	96.49/96.46	78.87	20	96.49/96.46	77.68	21
s13207.1	98.46/97.52	93.36	629100	97.66/97.52	93.51	32	97.66/97.52	93.11	31
s35932	89.81/89.81	36.68	20358	89.81/89.81	27.44	19	89.81/89.81	23.06	19

Top

5. CONCLUSION

In this paper, we have presented a novel and efficient test relaxation technique for combinational circuits. The technique is faster than the brute-force relaxation technique by several orders of magnitude. It is based on the critical path tracing (CRIPT) algorithm, and hence may result in a small drop in the fault coverage (if any) after relaxation. This is due to the approximate nature of CRIPT. Based on experimental results, a small drop in the fault coverage is observed for most of the circuits. Furthermore, the percentage of *x's* extracted is close to the one obtained by the brute-force test relaxation technique. Having a test relaxation technique is crucial for effective test compaction and compression. The applications of our test relaxation technique in improving the quality of test compaction and compression will be investigated in future work.

ACKNOWLEDGMENT

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

REFERENCES

1. M. Abramovici, M. Breuer and A. Friedman, 1990, *Digital System Testing and Testable Desig*n, IEEE Press.
2. A. Chandra and K. Chakrabarty, 2000, "Test Data Compression for System-On-a-Chip using Golomb Codes", *in Proc. of IEEE VLSI Test Symposiu*m, pp. 113-120.
3. A. Chandra and K. Chakrabarty, 2001, "Frequency-directed run-length (FDR) codes with application to system-on-a-chip test data compression ", *in Proc. IEEE VLSI Test Symposium*, pp. 42–47. J. Chang and C. Lin, 1995, "Test Set Compaction for Combinational Circuits", *IEEE Trans. on Computer Aided Desig*n, pp. 1370–1378.
4. A. El-Maleh, S. Zahir, and E. Khan, 2001, "A Geometric-Primitive-Based Compression Scheme for Testing Systems-on-a-Chip", *in Proc. IEEE VLSI Test Symposiu*m, pp. 54-59.
5. I. Hamzaoglu and J. Patel, 1998, "Test Set Compaction Algorithms for Combinational Circuits", *in Proc. International Conference on Computer-Aided Desig*n, pp. 283-289.
6. A. Jas and N. Touba, 1998, "Test Vector Decompression via Cyclical Scan Chains and Its Application to Testing Core-Based Designs", *in Proc. International Test Conferenc*e, pp. 458–464.
7. H. K. Lee and D. S. Ha, 1996, "HOPE: An Efficient Parallel Fault Simulator for Synchronous Sequential Circuits", *IEEE Trans. on Computer Aided Desig*n, vol. 15, no. 9, pp. 1048–1058.
8. M. Schulz, E. Trischhler, and T. Sarfert, 1998, "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System", *IEEE Trans. on Computer-Aided Desig*n, pp. 126–137.
9. Y. Zorian, E. J. Marinissen and S. Dey, 1998, "Testing Embedded-Core Based System Chips", *in Proc. International Test Conferenc*e, pp. 130–143.

Top