Test Vector Decomposition Based Static Compaction Algorithms for Combinational Circuits

Aiman H. El-Maleh and Yahya E. Osais
King Fahd University of Petroleum and Minerals

Testing system-on-chips involves applying huge amounts of test data, which is stored in the tester memory and then transferred to the chip under test 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 memory requirements for the tester. In this paper, a new approach to static compaction for combinational circuits, referred to as *test vector decomposition (TVD)*, is proposed. In addition, two new TVD based static compaction algorithms are presented. Experimental results for benchmark circuits demonstrate the effectiveness of the two new static compaction algorithms.

Categories and Subject Descriptors: B.6.2 [**Logic Design**]: Testing—*static compaction*; B.7.3 [**Integrated Circuits**]: Testing—*static compaction*; J.6 [**Computer-Aided Engineering**]: computer-aided design (CAD)

General Terms: Theory, Algorithms

Additional Key Words and Phrases: Static compaction, combinational circuits, taxonomy, test vector decomposition, independent fault clustering, class-based clustering

1. INTRODUCTION

Advances in the VLSI technology paved the way for System-on-Chips (SoCs). Traditional IC design, in which every circuit is designed from scratch and reuse is limited only to standard cell libraries, is more and more replaced by the SoC design methodology. However, this new design methodology has its own challenges. A major challenge is how to reduce the increasing volume of test data. Basically, there are two approaches: *compression* and *compaction*. In the first approach, test data is kept compressed while it is stored in the tester memory and transferred to the Chip Under Test (CUT). Then, it is decompressed on the CUT. This reduces the memory and transfer time requirements. In the second approach, however, the objective is to reduce the size of a test set while maintaining the same fault coverage.

Test compaction techniques are classified into two categories. The first category includes algorithms that can be integrated into the test generation process. Such algorithms are referred to as *dynamic* compaction algorithms. On the other hand,

Authors' addresses: Aiman H. El-Maleh, KFUPM, P.O. Box 1063, Dhahran 31261, Saudi Arabia; email: aimane@ccse.kfupm.edu.sa; Yahya E. Osais, KFUPM, P.O. Box 983, Dhahran 31261, Saudi Arabia; email: yosais@ccse.kfupm.edu.sa.

the second category includes algorithms that are applied after the test sets are generated. Such algorithms are referred to as *static* compaction algorithms. There are several approaches to static compaction of a given test set as will be shown in the next section.

Since test application time is proportional to the length of the test set that needs to be applied, it is desirable to apply shorter test sets that provide the same fault coverage at reduced test application time. Although static compaction algorithms do not typically produce test sets of sizes equal to those generated using dynamic compaction, interests in developing more efficient static compaction algorithms have increased [Miyase et al. 2002]. Static compaction has the following advantages over dynamic compaction. First, generating smaller test sets using dynamic compaction is time consuming because many attempts to modify partially specified test vectors to detect additional faults often fail [Miyase et al. 2002]. Secondly, dynamic compaction does not take advantage of random test pattern generation. Thirdly, static compaction is independent of ATPG.

Given a test set T with single stuck-at fault coverage FC_T for a combinational circuit, the static compaction problem can be formulated as to find another test set, T^*, for the same circuit such that $FC_{T^*} \geq FC_T$ and $|T^*| < |T|$ [Chang and Lin 1995]. It should be pointed out that in the above definition, there is no constraint on the individual fault coverage of each test vector and the proximity between the test vectors of T and T^*. That is, the fault coverage of each test vector needs not remain intact and T^* needs not be a subset of T.

This paper is structured as follows. First, we give a taxonomy of static compaction algorithms for combinational circuits. In this section, we review the existing static compaction algorithms and show how they fit in our taxonomy. Besides, we introduce and motivate the new concept of *test vector decomposition*. Then, we describe two new static compaction algorithms based on test vector decomposition. After that, we present and discuss the experimental results. Finally, we conclude by summarizing the results of the paper and their significance.

2. TAXONOMY OF STATIC COMPACTION ALGORITHMS

In this section, we give a taxonomy of static compaction algorithms for combinational circuits. We first start with an overview of the taxonomy. Then, we give a description of every class in the taxonomy with examples from the literature.

2.1 Overview

Static compaction algorithms for combinational circuits can be divided into three broad categories: (1) Redundant Vector Elimination, (2) Test Vector Modification, and (3) Test Vector Addition and Removal. Figure 1 shows our proposed taxonomy. In the first category, compaction is performed by dropping redundant test vectors. A redundant test vector is a vector whose faults are all detectable by other test vectors. Static compaction algorithms falling under this category can be further classified into two classes. The first class contains algorithms based on set covering in which faults are to be covered using the minimum possible number of test vectors. On the other hand, the second class contains algorithms based on test vector reordering in which reordering, fault simulation, fault distribution, and double detection are used to identify redundant test vectors and then drop them.

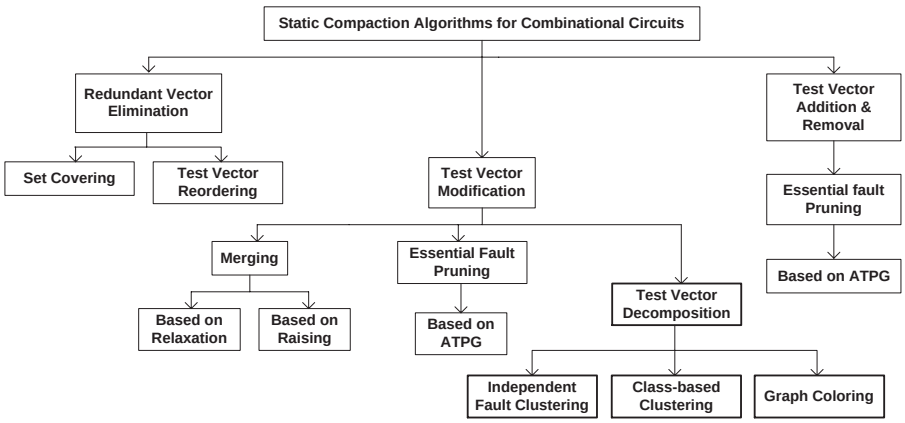

Fig. 1. Taxonomy of static compaction algorithms for combinational circuits.

In the second category, compaction is performed by modifying test vectors. Algorithms belonging to this category can be further classified into three classes. The first class contains algorithms based on merging of compatible test cubes. A test cube is a test vector that is partially specified. A test vector is made partially specified by unspecifying the unnecessary primary inputs. This process is referred to as relaxation. Relaxation can be performed using an ATPG or a stand-alone algorithm, such as [El-Maleh and Al-Suwaiyan 2002; Kajihara and Miyase 2001]. In addition to relaxation, raising can be used to enhance the compatibility among relaxed test vectors. If two relaxed test vectors conflict at one or more bit positions, they can be made compatible by raising one of them at the conflicting bit positions.

The second class contains algorithms that employ essential fault pruning to make some test vectors redundant. A test vector becomes redundant if it detects no essential faults. A fault is essential if it is detected only by a single test vector. Essential faults of a test vector can be pruned, i.e. made detected by some other test vectors, by reassigning values to those bits that are originally unspecified and have been randomly assigned values to detect additional faults.

The third class contains algorithms that are based on test vector decomposition. Test vector decomposition is the process of decomposing a test vector into its atomic components. An atomic component is a child test vector that is generated by relaxing its parent test vector for a single fault f. In this paper, we propose test vector decomposition as a new class of static compaction algorithms that modify test vectors to perform compaction.

Finally, the third category of static compaction algorithms consists of compaction algorithms that add new test vectors to a given test set in order to remove some of the already existing test vectors. The number of the newly added test vectors must be less than the number of test vectors to be removed. An ATPG is used to generate the new test vectors.

2.2 Set Covering

Test compaction for combinational circuits can be modeled as a set covering problem. The set cover is set up as follows. Each column of the detection matrix

Test	Faults
t_1	f_1,f_4
t_2	f_2,f_4
t_3	f_2,f_3

Fig. 2. Test vectors and their associated faults.

Fault	Test
f_1	t_1
f_2	t_2
f_3	t_3
f_4	t_1

Fig. 3. First test vector that detects every fault.

corresponds to a test vector and each row corresponds to a fault. If a test vector j detects fault i, then the entry (i, j) is one; otherwise, it is zero. In this setup, the total amount of memory required for building the detection matrix is $O(nf)$, where n is the number of test vectors and f is the number of faults.

Static compaction procedures based on set covering were described in [Flores et al. 1999; Boateng et al. 2001; Hochbaum 1996]. It should be pointed out that this approach has not been used much in the literature due to the huge memory and CPU time requirements.

2.3 Test Vector Reordering

Identification of redundant test vectors in a test set is an order dependent process. Given any order, redundant test vectors can be identified using fault simulation, fault distribution, or double detection. There are four variations of Test Vector Reordering (TVR) based static compaction algorithms.

2.3.1 *TVR with Fault Dropping Simulation.* Fault simulation of a test set in an order different from the order of generation is used as a fast and effective method to drop redundant test vectors. Under Reverse Order Fault simulation (ROF) [Schulz et al. 1988; Pomeranz and Reddy 2001], a test set is fault simulated with dropping in reverse order of generation. That is, a test vector that was generated later is fault simulated earlier. When it is simulated, a test vector that does not detect any new faults is removed from the test set.

The intuitive reason for this phenomenon is simply that test vectors which are further down the list detect faults which are most difficult to detect. Therefore, if we first fault simulate a test vector which is at the end of the list, it not only detects a hard fault right away, it also detects many others by pure chance. This way hard faults are out of the way early.

2.3.2 *TVR with Forward-Looking Fault Simulation.* The forward-looking fault simulation is an improved version of ROF [Pomeranz and Reddy 2001]. It is based on the idea that information about the first test vector that detects every fault can be used to drop test vectors that would not be dropped by ROF. That is, the yet undetected faults have lower indexed test vectors that detect them. So, some test vectors are skipped over and consequently dropped from the test set.

Let us consider the following example. Let the test set T be $\{t_1, t_2, t_3\}$ and the

fault set F be $\{f_1, f_2, f_3, f_4\}$. Figure 2 shows the test vectors with their associated faults and Figure 3 shows the first test vector that detects every fault. Conventional ROF first simulates t_3. This test will be retained in the test set to detect f_2 and f_3. Next, t_2 is simulated. Since it detects the new fault f_4, it is retained in the test set. Finally, t_1 is simulated and retained in the test set since it detects a new fault, which is f_1. No tests are dropped by ROF in this case.

Now, let us start ROF again taking into account the information given in Figure 3. ROF starts by simulating t_3. This test is retained in the test set to detect f_2 and f_3. Next, t_2 is simulated. t_2 detects the new fault f_4. However, f_4 is first detected by t_1. Therefore, we conclude that t_2 is not necessary for the detection of any yet undetected fault and we drop it from the test set. Finally, when t_1 is simulated, the remaining undetected faults f_1 and f_4 become detected and the detection process completes. In this case, one test vector is dropped from the test set.

2.3.3 *TVR with Fault Distribution.* In TVR with fault distribution, a test vector is fault simulated without fault dropping. Faults detected by every test vector are recorded. Besides, the number of test vectors that detect every fault is recorded. After that, given any order, a test vector whose number of essential faults is zero, i.e. the faults it detects can be distributed among other test vectors, is considered redundant and thus can be dropped. After a test vector is dropped, the number of test vectors that detect every one of its faults is reduced by one.

In [Hamzaoglu and Patel 1998], compaction based on fault distribution was used to compact test sets as a part of a dynamic compaction algorithm. The motive behind the proposed algorithm is the fact that ROF cannot identify a redundant test vector if some of the faults detected by it are only detected by the test vectors generated earlier. ROF can only identify a redundant test vector if all the faults detected by it are also detected by the test vectors generated later.

2.3.4 *TVR with Double Detection Fault Simulation.* Double Detection (DD) was first proposed in [Kajihara et al. 1995] as a dynamic compaction algorithm. Basically, when generating a new test vector, a yet undetected fault, called a primary target fault, is selected and a test vector t is generated to detect it. Next, other faults, called secondary target faults, are selected one at a time and the unspecified values in t are specified appropriately to detect the secondary target faults until no unspecified values remain in t or no additional secondary target faults can be detected. In choosing the secondary target faults, faults that are not detected are first considered and then faults detected at most once by earlier generated test vectors are considered. Faults are dropped from the list of target faults when they are detected twice. Test vectors that detect faults which are detected only once, i.e. essential test vectors, are marked. After the test generation is complete (when all the faults are detected at least once or aborted or proved to be untestable), the following static compaction procedure is used to reduce the test set size. The generated test vectors are fault simulated with dropping in the following order. First, all the essential test vectors are simulated in the order they were generated. The essential test vectors are followed by the non-essential test vectors in the order opposite to the order in which they were generated. During the fault simulation process, a test vector that does not detect any new fault is dropped. It should be

Table I. Definition of the merging operation.

$\circ$	0	1	x
0	0	ϕ	0
1	ϕ	1	1
x	0	1	x

pointed out that essential test vectors cannot be dropped and thus simulating them first maximizes the ability to drop other test vectors.

DD can be used in static compaction procedures, e.g. [Lin et al. 2001]. However, since most test generators do not attempt to target faults for a second detection and do not use non-fault dropping simulation, they do not collect all the information necessary for static compaction based on DD. Therefore, the necessary information must be collected in a preprocessing step.

2.4 Merging

Static compaction algorithms in this class can be divided into two groups. The first group contains algorithms based on the very simple and efficient approach of merging compatible test cubes. A test cube is a relaxed test vector. A test vector is relaxed by unspecifying the unnecessary primary inputs. A test vector can be relaxed using an ATPG or a stand-alone algorithm, such as [El-Maleh and Al-Suwaiyan 2002; Kajihara and Miyase 2001]. A merging procedure employing relaxation proceeds as follows. Given a test set T, test vectors in T are all relaxed. Then, an iterative search is performed for pairs of compatible test vectors. Two test vectors t_i and t_j are compatible if they do not specify complementary values in any bit position. If any two vectors, say t_i and t_j, are compatible, they are replaced by the vector $t_i \circ t_j$, where $\circ$ represents the merging operation (see the definition in Table I). The new test vector $t_i \circ t_j$ has all the binary values of both t_i and t_j. Hence, by a repetitive application of the above compaction operation, many test vectors (two or more) can be combined into fewer test vectors. As a result the total number of test vectors that need to be applied with the same fault detection capabilities is reduced. Examples of this approach can be found in [El-Maleh and Al-Suwaiyan 2002; Ayari and Kaminska 1994; Miyase et al. 2002].

In the second group, algorithms employ in addition to the relaxation operation a *raising* operation. For a test vector t, the raising operation $raise(t, i)$ tries to set the i^{th} bit of t to x while preserving the coverage of the essential faults of t. The raising operation was proposed in [Chang and Lin 1995]. Raising is used to enhance compatibility among relaxed test vectors. For example, if two relaxed test vectors, say t_i and t_j, conflict at one or more bit positions, they can be made compatible by raising one of them at the conflicting bit positions. Typically, raising is used to resolve conflicts when a test set contains no compatible test vectors.

2.5 Test Vector Decomposition

Test Vector Decomposition (TVD) is the process of decomposing a test vector into its atomic components. An atomic component is a child test vector that is generated by relaxing its parent test vector for a single fault f. That is, the child test vector contains the assignments necessary for the detection of f. Besides, the child test vector may detect other faults in addition to f. For example, consider the test vector

t_p = 010110 that detects the set of faults $F_p = \{f_1, f_2, f_3\}$. Using the relaxation algorithm in [El-Maleh and Al-Suwaiyan 2002], t_p can be decomposed into three atomic components, which are (f_1,01xxxx), (f_2,0x01xx), and (f_3,x1xx10). Every atomic component detects the fault associated with it and may accidentally detect other faults. An atomic component cannot be decomposed any further because it contains the assignments necessary for detecting its fault.

Static compaction based on merging is a very simple and efficient technique. However, it has the following problems. First, for a highly incompatible test set, merging achieves little reduction. Secondly, raising is a costly operation. Thirdly, a test vector must be processed as a whole. Therefore, we propose that a test vector be decomposed into its atomic components before it is processed. In this way, a test vector that is originally incompatible with all other test vectors in a given test set can be eliminated if its components can be merged with other test vectors.

By decomposing a test vector into its atomic components, a merging based compaction algorithm will have more degree of freedom. This is because of the fact that the number of unspecified bits in an atomic component is much larger than that in a parent test vector. Thus, the probability of merging a component is higher than that of merging its parent test vector.

The problem of static compaction based on TVD can be modeled as a graph coloring problem. Basically, given a test set T with single stuck-at fault coverage FC_T, the set of atomic components C_T is first obtained. Then, a graph G is built. In G, every node corresponds to a component and an edge exists between two nodes if their corresponding components are incompatible. Now, our objective is to partition C_T into k subsets such that k is as small as possible and no adjacent nodes belong to the same subset. The fault coverage of the new test set T^* whose size is k should be greater than or equal to FC_T.

It is well known that graph coloring is an NP-hard problem [Garey and Johnson 1979]. Thus, efforts of researchers are devoted to heuristic methods, rather than exact ones. Heuristic methods are simple schemes in which nodes are colored sequentially according to some criteria.

2.6 Essential Fault Pruning

Generally speaking, pruning a fault of a test vector decreases the number of its faults by one. A test vector becomes redundant if all of its faults are pruned. Fault Pruning (FP) is implemented as follows. Given a test vector t, an attempt is made to detect each of its faults by modifying the other test vectors in the test set. A fault of t is said to be pruned if it becomes detected by another test vector after the modification. If all the faults of t are pruned, then t can be removed from the test set.

The above operation of modifying a test vector, say $t^{'}$, to further detect an additional fault f of another test vector t is basically achieved by generating a new test vector $t^{''}$ such that $\mathrm{DET}(t^{''}) = \mathrm{DET}(t^{'}) \cup f$, where $\mathrm{DET}(t)$ is the set of faults detected by t. Multiple Target Faults Test Generation (MTFTG) is used for this purpose. In MTFTG, a test vector is to be found for a set of target faults. MTFTG will fail if there exists at least two independent faults in the set of target faults. Two faults are independent if they cannot be detected by a single test vector.

The run time of an FP based static compaction procedure can be greatly improved

by considering only essential faults. A fault is defined to be an essential fault of a test vector t if it is detected only by t. The set of essential faults of t is denoted by ESS(t). It should be pointed out that whenever a test vector t is eliminated, for every fault belonging to the set DET(t) - ESS(t), the number of test vector detecting it is reduced by one.

Few FP based static compaction algorithms have been reported in the literature. Generally, they fall into two categories. In the first category, a test vector is modified such that it detects the new additional faults. The test vector already detects its essential faults. Therefore, the test generation time for the essential faults is eliminated. Examples of such static compaction algorithms can be found in [Hamzaoglu and Patel 1998; Chang and Lin 1995; Reddy et al. 1992; Hamzaoglu and Patel 2000]. On the other hand, in the second category, a set of N test vectors is replaced by a set of $M < N$ new test vectors. The basic idea is to determine the faults that are detected only by one or more test vectors among the N test vectors to be replaced and find $M < N$ test vectors that detect all theses faults. Examples of such static compaction algorithms can be found in [Kajihara et al. 1995; 1994].

3. TEST VECTOR DECOMPOSITION BASED STATIC COMPACTION ALGORITHMS

3.1 Independent Fault Clustering

3.1.1 *Preliminaries.* Independent faults were defined in [Akers and Krishnamurthy 1989]. Basically, given a combinational circuit, let T_i be the set of all possible test vectors that detect f_i and T_j be the set of all possible test vectors that detect f_j. Then, two faults f_i and f_j are independent if and only if $T_i \cap T_j = \phi$. Independence among faults can also be defined with respect to a test set T. Let T_i' be the set of test vectors in T that detect f_i and T_j' be the set of test vectors in T that detect f_j. Then, two faults f_i and f_j are independent with respect to T if and only if $T_i' \cap T_j' = \phi$. In this paper, we use the term independent faults to mean independent faults with respect to a test set.

A fault set is called an Independent Fault Set (IFS) if all the faults in this set are pairwise independent. The problem of computing a maximum size IFS is NP-Hard [Krishnamurthy and Akers 1984]. Therefore, only *maximal* IFSs can be practically computed. Heuristic methods for computing IFSs were described in [Akers and Joseph 1987; Akers and Krishnamurthy 1989; Tromp 1991; Pomeranz and Reddy 1992].

IFSs were used in [Akers and Joseph 1987; Tromp 1991; Pomeranz et al. 1993; Kajihara et al. 1994; 1995; Chang and Lin 1995; Wang and Stabler 1995; Hamzaoglu and Patel 1998; 2000]. The importance of independent faults is threefold [Pomeranz and Reddy 1992]. First, they provide a lower bound on the size of the minimum test set, thus making it possible to estimate the success of test pattern generators in generating small test sets. Secondly, independent faults provide a method for ordering target faults for test generation. Ordering has been shown to be important for obtaining small test sets and reducing test generation time [Pomeranz et al. 1993]. Thirdly, the use of independent faults improves the efficiency of static compaction algorithms based on essential fault pruning.

Algorithm IFC
Input: A test set T of size $|T|$ and fault coverage FC_T.
Output: A new test set T^* such that $|T^*| \leq |T|$ and $FC_{T^*} \geq FC_T$.

1. Fault simulate T without fault dropping.
2. For every essential fault f that is detected by a test vector t:
   2.1. Extract the atomic component c_f from t.
   2.2. If the number of compatibility sets is zero, create a new compatibility set, map c_f to it, and then go to Step 2.
   2.3. Map c_f to an existing compatibility set, if possible, and then go Step 2.
   2.4. Create a new compatibility set and map c_f to it.
3. Find sets of independent faults.
4. Sort sets of independent faults in decreasing order of their sizes.
5. For every fault in an IFS, sort the test vectors that detect the fault in decreasing order of the number of faults they detect.
6. For every fault f, where f belongs to an IFS:
   6.1. For every test vector t that detects f:
      6.1.1. Extract the atomic component c_f from t.
      6.1.2. If the number of compatibility sets is zero, create a new compatibility set, map c_f to it, and then go to Step 6.
      6.1.3. Map c_f to an existing compatibility set, if possible, and go to Step 6.
   6.2. Create a new compatibility set and map c_f to it.
7. Return T^*.

Fig. 4. The IFC algorithm.

3.1.2 *Algorithm Description.* In Independent Fault Clustering (IFC) algorithms, IFSs are first derived. Then, a fault matching procedure is used to find sets of compatible faults, i.e. faults that can be detected by a single test vector. In the IFS derivation phase, independent faults are identified with respect to a test set. On the other hand, in the fault matching phase, compatible components, corresponding to compatible faults, are mapped to the same compatibility set. Whenever a component is mapped to a compatibility set, it is merged with the partial test vector of that compatibility set. At the end, every compatibility set represents a single test vector.

Our IFC algorithm is shown in Figure 4 and proceeds as follows. First, the given test set T is fault simulated without fault dropping. This step is performed to find the number and set of test vectors that detect every fault. Secondly, essential faults are matched. In this step, for every essential fault f detected by t, the atomic component c_f corresponding to f is extracted from t. Then, for every compatibility set CS_i, if c_f is compatible with the partial test vector in CS_i, c_f is mapped to CS_i. On the other hand, if the number of compatibility sets is zero or c_f is incompatible with all partial test vectors in the existing compatibility sets, a new compatibility set is created and c_f is mapped to it.

It should be observed that an essential fault has a single component while non-essential faults have more than one. Therefore, if a component of a non-essential fault f is incompatible with all the partial test vectors in the existing compatibility sets, the other components of f will be tried before creating a new compatibility set. On the other hand, if the component of an essential fault is incompatible with all the partial test vectors in the existing compatibility sets, a new compatibility

Table II. Example test vectors and their components.

Test Vector		*Fault Detected*	*Fault Component*
v_1	$0xx11x0x10$	f_1	$xxx1xx0xxx$
		f_2	$0xxx1xxxx0$
		f_3^e	$0xxxxx0x10$
v_2	$10x1xxxx00$	f_1	$x0x1xxxxxx$
		f_4^e	$1xxxxxx00$
		f_5^e	$10x1xxxxxx$
v_3	$0xx0xxx00x$	f_6^e	$0xx0xxxxxx$
		f_7^e	$xxxxxx00x$
v_4	$111xxxx0x0$	f_2	$x11xxxxxx0$
		f_8^e	$11xxxxx0xx$
v_5	$xx000x11x1$	f_9	$xx000xxxxx$
		f_{10}^e	$xx0xxx11x1$
v_6	$x0x01xxx1x$	f_9	$xxx0xxxx1x$
		f_{11}^e	$x0x01xxxxx$

set must be created. Hence, essential faults should be matched first. Another advantage of first matching essential faults is that the number of faults that will be considered when deriving IFSs is reduced.

After essential faults are matched, IFSs are derived. Faults in an IFS are pairwise independent. Therefore, a fault f_i can be added to an IFS S if and only if for every fault f_j in S, the intersection of the sets of test vectors that detect f_i and f_j is empty. Next, IFSs are sorted in decreasing order of their sizes and for every fault in an IFS, the set of test vectors that detect the fault is sorted in decreasing order of the number of faults they detect. This is because a component that is extracted from a test vector that detects a large number of faults has high compatibility since it is compatible with all the components of the faults detected by that test vector.

Next, for every fault f in an IFS, its atomic component is extracted and then mapped to an appropriate compatibility set. For every component of a fault f, if it is incompatible with all partial test vectors in the existing compatibility sets, a new component will be tried. A new compatibility set is created if the number of compatibility sets is zero or all components of a fault f are incompatible with all partial test vectors in the existing compatibility sets. At the end, the algorithm returns the number of compatibility sets as the size of the new test set.

3.1.3 *Illustrative Example.* Table II shows an example of six test vectors and the faults they detect along with the components required for detecting the faults. The superscript e attached to some faults indicates that the faults are essential. As can be seen from the table, the six vectors cannot be merged together as there is at least one bit in conflict between each vector pair. Thus, the test vector merging method cannot compact these test vectors.

Table III illustrates applying the IFC algorithm on the test vectors in Table II. The first three columns show the clusters created after mapping the components of essential faults. After essential faults are mapped, IFSs are created and then their faults are mapped. There are two IFSs, namely $IFS_1 = \{f_1, f_9\}$ and $IFS_2 = \{f_2\}$. Columns four and five show the clusters after mapping the components of faults in the IFSs. Finally, the last column shows the compacted test vectors after merging

Table III. Example test vectors and their components.

	$AfterMapping$ $EssentialFaults$		$AfterMapping$ $IndependentFaultSets$		$AfterMerging$ $Components$
$Cluster$	$Fault$	$Fault\,Component$	$Fault$	$Fault\,Component$	$Test\,Vector$
1	f_3	$0xxxxx0x10$	f_3	$0xxxxx0x10$	$00x01x0x10$
	f_6	$0xx0xxxxxx$	f_6	$0xx0xxxxxx$	
	f_{11}	$x0x01xxxxx$	f_{11}	$x0x01xxxxx$	
			f_2	$0xxx1xxxx0$	
2	f_4	$1xxxxxxx00$	f_4	$1xxxxxxx00$	$10x1xx0000$
	f_5	$10x1xxxxxx$	f_5	$10x1xxxxxx$	
	f_7	$xxxxxxx00x$	f_7	$xxxxxxx00x$	
			f_1	$xxx1xx0xxx$	
3	f_8	$11xxxxx0xx$	f_8	$11xxxxx0xx$	$11000xx0xx$
			f_9	$xx000xxxxx$	
4	f_{10}	$xx0xxx11x1$	f_{10}	$xx0xxx11x1$	$xx0xxx11x1$

Algorithm Iter_IFC
Input: A test set T of size $|T|$ and fault coverage FC_T.
Output: A new test set T^* such that $|T^*| \leq |T|$ and $FC_{T^*} \geq FC_T$.

1. Randomly fill the unspecified bits in T.
2. T^* = IFC(T)
3. If $|T^*| < |T|$, copy T^* to T and go to Step 1.
   Else If $|T^*| == |T|$, return T^*.
   Else return T.

Fig. 5. The iterative IFC algorithm.

the components in each cluster. Since the number of clusters obtained is four, the compacted test set is of size four. Hence, two test vectors were eliminated.

3.1.4 *Iterative IFC.* The level of compaction achievable by our IFC algorithm can be improved in two ways. First, after a component is generated for a fault, the component is fault simulated and the faults detected by it are marked as detected. In this way, a large portion of the faults will not be considered subsequently since they are already detected. Based on our experimental investigations, we noticed that this extra step increases the runtime and improves the results very little. Secondly, the IFC algorithm can be called on a test set iteratively. Basically, the new test set generated is treated as the test set to be compacted. Therefore, IFC is carried out iteratively until the length of the test set cannot be reduced any more. This process is called *iterative* IFC and is shown in Figure 5. Unspecified bits in the test set T are assigned random values before every call to the IFC algorithm.

It should be pointed out that any static compaction algorithm can be used after our IFC algorithm. In fact, given a test set T, the IFC algorithm will generate a new test set T^* whose characteristics are different from the characteristics of T. Thus, a static compaction algorithm that cannot compact T may manage to compact T^*.

3.2 Class-based Clustering

3.2.1 *Preliminaries.* Given the set of components of every test vector in a test set, a test vector can be eliminated if its components can be all moved to other test vectors. Moving a component to a test vector is implemented by merging the

component with the destination test vector. Even though the idea is very simple, it is not always possible to move a component to a new test vector. This is because of two problems: (1) blocking and (2) conflicting components. In the former, a component c_i is blocked from being moved to a test vector t when it becomes incompatible with it. c_i becomes incompatible with t when another component c_j that is incompatible with c_i is moved to t. In the latter, however, a test vector is uneliminatable if it contains at least one conflicting component. A conflicting component cannot be moved to any other test vector in the given test set.

Definition 1. (Conflicting Component). A component c of a test vector t belonging to a test set T is called a Conflicting Component (CC) if it is incompatible with every other test vector in T.

The number of CCs in a test vector determines its degree of hardness. The degree of hardness of a test vector is basically a measure of how much hard a test vector is to eliminate. Test vectors can be classified based on their degree of hardness.

Definition 2. (Degree of Hardness of a Test Vector). A test vector is at the n^{th} degree of hardness if it has n CCs.

Definition 3. (Class of a Test Vector). A test vector belongs to class k if its degree of hardness is k.

A CC can be moved to a test vector t if the characteristics of t are changed. That is, a CC c_i is movable to a test vector t, if the components in t incompatible with c_i can be moved to other test vectors. The set of test vectors to which c_i can be moved is referred to as the set of candidate test vectors of c_i. A test vector whose CCs are all movable is referred to as a potential test vector.

Definition 4. (Movable CC). Let c_i be a CC in a test vector t_s, β be a set of components in a test vector t_d such that c_i is incompatible with every component c_j in β, S_j be the set of test vectors compatible with c_j. Then, c_i is movable to t_d if and only if $S_j \neq \phi$ for every c_j in β.

Definition 5. (Set of Candidate Test Vectors of a CC). The set of candidate test vectors of a CC c_i, denoted by $S_{cand}(c_i)$, contains all test vectors to which c_i can be moved.

Definition 6. (Potential Test Vector). Let α be the set of CCs in a test vector t. t is a potential test vector that belongs to class $|\alpha|$ if and only if for every CC c_i in α, c_i is movable.

3.2.2 *Algorithm Description.* After stating the necessary definitions, we now describe our Class-Based Clustering (CBC) algorithm. The CBC algorithm is based on the idea of dividing test vectors into classes and then heuristically processing test vectors of every class. A test vector is eliminated if its components can be all moved to other test vectors. Eventually, in the final test set, every test vector represents a cluster whose components originally belong to test vectors in different classes. This is why the technique is called CBC.

The CBC algorithm is shown in Figure 6 and proceeds as follows. First, the given test set is fault simulated without fault dropping. This step is performed

Algorithm CBC
Input: A test set T of size $|T|$ and fault coverage FC_T.
Output: A new test set T^* such that $|T^*| \leq |T|$ and $FC_{T^*} \geq FC_T$.

1. Fault simulate T without fault dropping.
2. Sort test vectors in increasing order of their number of faults.
3. Generate atomic components (see Figure 7).
4. Sort test vectors in decreasing order of their number of components.
5. Remove redundant components using fault dropping simulation.
6. For every test vector, merge its components together.
7. Classify test vectors.
8. Process class zero test vectors (see Figure 8).
9. For every test vector, merge its components together.
10. Reclassify test vectors.
11. Process class one test vectors (see Figure 9).
12. For every test vector, merge its components together.
13. Reclassify test vectors.
14. Process class i test vectors, where $i > 1$ (see Figure 11).

Fig. 6. The CBC algorithm.

Algorithm Gen_Comp
1. For every test vector t:
   1.1. For every fault f detected by t:
      1.1.1. If f is *essential*:
         a. Extract the atomic component c_f from t.
      Else
         b. Decrement the number of test vectors detecting f by one.

Fig. 7. Algorithm for generating components.

to find the number and set of test vectors that detect every fault. Secondly, test vectors are sorted in increasing order of their number of faults. Then, atomic components of test vectors are generated. Component generation is performed such that components are extracted from essential test vectors. An essential test vector is a test vector that detects at least one essential fault. The component generation algorithm is shown in Figure 7 and proceeds as follows. For every fault f detected by t, if the number of test vectors that detect f is one, i.e. f is an essential fault, the component of f is extracted from t; otherwise, the number of test vectors that detect f is reduced by one. Therefore, a test vector that detects no essential faults is eliminated. The sorting step preceding component generation improves the number of eliminated test vectors. Note that a component of a fault is extracted from a test vector that detects a large number of faults.

After obtaining the set of components of every test vector, test vectors are sorted in decreasing order of their number of components. This helps maximize the number of redundant components. Redundant components are dropped using fault simulation with dropping. After that, every test vector is reconstructed by merging its components together. Then, test vectors are classified and processed.

Class zero test vectors are processed as shown in Figure 8. First, test vectors are

Algorithm Proc_Class_0_TVs

1. Sort class zero test vectors in increasing order of their number of components.
2. For every class zero test vector, compute its blockage value.
3. For every class zero test vector t whose blockage value is zero:
   3.1. Move components of t to appropriate test vectors.
   3.2. Eliminate t.
   3.3. Update S_{comp} of components belonging to other class zero test vectors.
   3.4. Update the blockage values of other class zero test vectors.
4. Sort class zero test vectors in increasing order of their number of components.
5. For every remaining class zero test vector t:
   5.1. If components of t can be all moved:
      5.1.1. Move components of t to appropriate test vectors.
      5.1.2. Eliminate t.
      5.1.3. Update S_{comp} of components belonging to other class zero test vectors.

Fig. 8. Algorithm for processing class zero test vectors.

sorted in increasing order of their number of components. This way a test vector with a small number of components has a higher chance of getting eliminated. After that, for every test vector, its blockage value is computed. The blockage value of a test vector t, denoted by *TVB(t)*, can be defined as the sum of the blockage values of the individual components making up t. This can be shown mathematically as follows.

$$TVB(t) = \sum_{i=1}^{NumComp} CB(c_i),$$

where $CB(c_i)$ is the blockage value of component c_i belonging to the set of components of t and $NumComp$ is the number of components making up t.

$CB(c_i)$ is mathematically defined as follows.

$$CB(c_i) = Min\{CB(c_i, t_j)\},$$

where $CB(c_i, t_j)$ is the number of class zero test vectors that will be blocked when component c_i is moved to test vector t_j, t_j belongs to $S_{comp}(c_i)$, and $S_{comp}(c_i)$ is the set of test vectors compatible with c_i. Note that when computing $CB(c_i, t_j)$ only components $c_k \in t_j$ such that $S_{comp}(c_k) = 1$ and c_k is in conflict with c_i needs to be considered.

Components of a test vector whose blockage value is zero can be moved without blocking any class zero test vector. Therefore, for any class zero test vector whose blockage value is zero, its components are moved to appropriate test vectors and then it is eliminated. A component c_i is moved to a test vector t_j in $S_{comp}(c_i)$ such that $CB(c_i, t_j) = 0$. If there is more than one test vector, a test vector with the smallest number of components is selected. This is based on the assumption that a test vector with a small number of components has a smaller probability of

Table IV. Example test vectors showing that although v_1 has a zero blockage value, it blocks v_2.

Test Vector	*Component*	*Set of Compatible Test Vectors*
v_1	c_{11}	$\{v_3\}$
	c_{12}	$\{v_4\}$
v_2	c_{21}	$\{v_3, v_4\}$

conflicts with other components. The blockage values of the other class zero test vectors must be updated after merging the components of a class zero test vector. Note that the blockage value of a class zero test vector t needs to be updated if t has at least one component c_i whose S_{comp} has been modified or t receives new components. Besides, the blockage value needs to be updated if t has at least one component c_i in conflict with another component c_j such that $S_{comp}(c_j)$ has been modified and $S_{comp}(c_j) = 1$.

Next, remaining class zero test vectors, having non-zero blockage value, are sorted in increasing order of their number of components. A remaining test vector t can be eliminated if for every component c_i in t, $S_{comp}(c_i) \neq \phi$. A component is heuristically moved to a test vector with the smallest number of components. S_{comp} of every component must be updated after eliminating every test vector.

It is worth mentioning that the technique we use for computing the blockage value of a class zero test vector is not exact. Consider for example the two test vectors shown in Table IV. Both vectors are in class zero. Suppose that c_{21} is in conflict with both c_{11} and c_{12}. Our technique will compute the blockage value of v_1 as zero although it will block v_2. The correct blockage value of v_1 is one.

After processing class zero test vectors, every test vector is reconstructed by merging its components together. Then, test vectors are reclassified. After that, class one test vectors are processed as shown in Figure 9. Basically, for every class one test vector, S_{cand} of the CC is found and potential test vectors are marked. Then, for every test vector in S_{cand}, the number of class one potential test vectors whose CCs can be moved to it is found. Besides, test vectors in S_{cand} of every class one potential test vector are sorted according to their types, i.e. a non-potential test vector should come before a potential test vector. If two test vectors have the same type, they are sorted in decreasing order of the number of CCs that can be moved to every one of them.

After processing the S_{cand} of the CC of every class one potential test vector, class one potential test vectors are sorted in decreasing order of the number of non-potential test vectors in S_{cand}. This is done to reduce the number of CCs that may be moved to potential test vectors. After that, for every class one potential test vector t_p^1, its CC is moved to a test vector selected from S_{cand}, call it t_d, remaining components making up t_p^1 are moved to appropriate test vectors, and test vectors are reclassified (see Figure 10). Before moving a remaining component, test vectors in its S_{comp} are sorted in decreasing order of their degree of hardness. This is to avoid increasing the number of components of test vectors having lower degrees of hardness since they have better chances of getting eliminated. After t_p^1 is eliminated, for every test vector t_p^2 whose CC can be moved to t_d, t_p^2 is processed in the same way as t_p^1.

Algorithm Proc_Class_1_TVs

1. For every class one test vector t:
   1.1. Find S_{cand} of the CC.
   1.2. If $S_{cand} \neq \phi$, mark t as potential.
2. For every class one potential test vector t:
   2.1. For every test vector in S_{cand}, find the number of class one potential test vectors whose CCs can be moved to it.
   2.2. Sort test vectors in S_{cand} according to their types.
3. Sort class one potential test vectors in decreasing order of the number of non-potential test vectors in S_{cand}.
4. For every unprocessed class one potential test vector t_p^1:
   4.1. Merge t_p^1 (see Figure 10). Denote by t_d the test vector to which the CC of t_p^1 has been moved.
   4.2. If t_p^1 has been eliminated, then for every class one potential test vector t_p^2 whose CC can be moved to t_d, merge t_p^2.

Fig. 9. Algorithm for processing class one test vectors.

Algorithm Merge_Class_1_Potential_TV
Input: A class one potential test vector t_p.

1. If the CC in t_p is movable:
   1.1. Move the CC to an appropriate test vector selected from S_{cand}.
   1.2. Move the remaining components to appropriate test vectors.
2. Reclassify test vectors.

Fig. 10. Algorithm for merging class one potential test vectors.

After processing class one test vectors, test vectors are reconstructed and then reclassified. Next, test vectors in class i, where $i > 1$, are processed as shown in Figure 11. Basically, for every class, if the number of potential test vectors is greater than zero, potential test vectors are marked. Then, if all the CCs of a potential test vector t can be moved, t is marked eliminated and its components are moved to other test vectors. If at least one CC of t cannot be moved, t is skipped. Several heuristics can be tried when moving a component. In our case, before moving a CC c_i, test vectors in $S_{cand}(c_i)$ are sorted in decreasing order of the number of components incompatible with c_i. Besides, before moving a component c_j that is not CC, test vectors in $S_{comp}(c_i)$ are sorted in decreasing order of their degree of hardness. Note that test vectors are reclassified after moving every CC and set of remaining components.

3.2.3 *Illustrative Example.* We now illustrate the application of the CBC algorithm on the example given in Table II. Table V shows the test vectors and their components after the component generation phase. In the example, redundant components are not dropped for simplicity. The second column shows the class of each test vector. The third and fourth columns show the faults detected by each test vector and their components, respectively. The last column shows the set of compatible test vectors with each component.

We first consider test vectors in class zero with the smallest number of compo-

Algorithm Proc_Remaining_Classes

1. For every class i, where $i > 1$:
   1.1. Find the set of class i potential test vectors.
   1.2. For every class i potential test vector t_p:
      1.2.1. For every CC in t_p:
         a. Move the CC to an appropriate test vector; otherwise, go to Step 1.2.
         b. Reclassify test vectors.
      1.2.2. If all CCs in t_p have been moved:
         a. Move remaining components.
         b. Reclassify test vectors.

Fig. 11. Algorithm for processing remaining classes.

Table V. Test vectors and their generated components.

Test Vector	*Class*	*Fault*	*Component*	*Set of Compatible Test Vectors*
v_1	0	f_3	$0xxxx0x10$	$\{v_6\}$
v_2	0	f_1	$x0x1xxxxxx$	$\{v_1, v_5\}$
		f_4	$1xxxxxx00$	$\{v_4\}$
		f_5	$10x1xxxxx$	$\{v_5\}$
v_3	0	f_6	$0xx0xxxxx$	$\{v_1, v_5, v_6\}$
		f_7	$xxxxxx00x$	$\{v_2, v_4\}$
v_4	1	f_2	$x11xxxxx0$	$\{v_1, v_3\}$
		f_8	$11xxxx0xx$	$\{\}$
v_5	0	f_{10}	$xx0xxx11x1$	$\{v_6\}$
v_6	0	f_9	$xxx0xxxx1x$	$\{v_1, v_4, v_5\}$
		f_{11}	$x0x01xxxx$	$\{v_1, v_3, v_5\}$

Table VI. Test vectors and their components after eliminating v_1.

Test Vector	*Class*	*Fault*	*Component*	*Set of Compatible Test Vectors*
v_2	0	f_1	$x0x1xxxxxx$	$\{v_5\}$
		f_4	$1xxxxxx00$	$\{v_4\}$
		f_5	$10x1xxxxxx$	$\{v_5\}$
v_3	0	f_6	$0xx0xxxxx$	$\{v_5, v_6\}$
		f_7	$xxxxxx00x$	$\{v_2, v_4\}$
v_4	1	f_2	$x11xxxxx0$	$\{v_3\}$
		f_8	$11xxxx0xx$	$\{\}$
v_5	1	f_{10}	$xx0xxx11x1$	$\{\}$
v_6	1	f_9	$xxx0xxxx1x$	$\{v_4, v_5\}$
		f_{11}	$x0x01xxxx$	$\{v_3, v_5\}$
		f_3	$0xxxx0x10$	$\{\}$

nents. We choose to eliminate v_1 by moving its component to v_6. Table VI shows the test vectors and their components after eliminating v_1. We next eliminate v_3 by moving the component of f_6 to v_6 and the component of f_7 to v_4. Table VII shows the test vectors and their components after eliminating v_3. We next eliminate v_2 by moving the components of f_1 and f_5 to v_5 and the component of f_4 to v_4. Table VIII shows the test vectors and their components after eliminating v_2.

Table VII. Test vectors and their components after eliminating v_3.

Test Vector	*Class*	*Fault*	*Component*	*Set of Compatible Test Vectors*
v_2	0	f_1	$x0x1xxxxxx$	$\{v_5\}$
		f_4	$1xxxxxx00$	$\{v_4\}$
		f_5	$10x1xxxxxx$	$\{v_5\}$
v_4	2	f_2	$x11xxxxx0$	$\{\}$
		f_8	$11xxxx0xx$	$\{\}$
		f_7	$xxxxxx00x$	$\{v_2\}$
v_5	1	f_{10}	$xx0xxx11x1$	$\{\}$
v_6	1	f_9	$xxx0xxxx1x$	$\{v_5\}$
		f_{11}	$x0x01xxxxx$	$\{v_5\}$
		f_3	$0xxxxx0x10$	$\{\}$
		f_6	$0xx0xxxxxx$	$\{v_5\}$

Table VIII. Test vectors and their components after eliminating v_2.

Test Vector	*Class*	*Fault*	*Component*	*Set of Compatible Test Vectors*
v_4	4	f_2	$x11xxxxx0$	$\{\}$
		f_8	$11xxxx0xx$	$\{\}$
		f_7	$xxxxxx00x$	$\{\}$
		f_4	$1xxxxxx00$	$\{\}$
v_5	3	f_{10}	$xx0xxx11x1$	$\{\}$
		f_1	$x0x1xxxxxx$	$\{\}$
		f_5	$10x1xxxxxx$	$\{\}$
v_6	4	f_9	$xxx0xxxx1x$	$\{\}$
		f_{11}	$x0x01xxxxx$	$\{\}$
		f_3	$0xxxxx0x10$	$\{\}$
		f_6	$0xx0xxxxxx$	$\{\}$

At this stage, none of the test vectors can be eliminated. So, the resulting compacted test set is obtained by merging the components in each test vector. The final test set is of size three and is {111xxxx000,1001xx11x1,00x01x0x10}.

3.3 Worst-Case Analysis

We analyze here the worst-case storage and runtime requirements of our algorithms. In the analysis, we assume that the test set, fault list, and circuit structure are given as inputs. Therefore, their memory and time requirements are not considered. Throughout the analysis, the number of test vectors in a test set will be denoted by N_T, size of a test vector will be denoted by N_{PI}, and the number of faults and gates in a circuit will be denoted by N_F and N_G, respectively.

3.3.1 *Space Complexity.* The space complexity of the IFC algorithm is analyzed as follows. In Step 1, a memory space of size $O(N_F N_T)$ is required for storing the indexes of test vectors detecting every fault. In Step 2, a memory space of size $O(N_{PI})$ is required for storing a component when it is generated. Besides, a memory space of size $O(N_T N_{PI})$ is required for storing test vectors of compatibility sets. In Step 3, the memory space required for building IFSs is $O(N_F)$. Finally, in Step 6, a memory space of size $O(N_{PI})$ is required for storing a component when it

is generated. Besides, a memory space of size $O(N_T N_{PI})$ is required for storing test vectors of compatibility sets. Hence, the IFC algorithm has the space complexity $O(N_T(N_F + N_{PI}))$.

The space complexity of the CBC algorithm is analyzed as follows. In Step 1, a memory space of size $O(N_F N_T)$ is required for storing the set of faults detected by every test vector. In Step 3, a memory space of size $O(N_F N_{PI})$ is required for storing all the components. For every component, a list of size $O(N_T)$ is required for storing the indexes of compatible/candidate test vectors. The complexity of this step is $O(N_F N_T)$. Furthermore, for every component belonging to class zero test vector, a list of size $O(N_T)$ is required for storing the blockage values. The complexity of this step is $O(N_F N_T)$. Hence, the CBC algorithm has the space complexity $O(N_F(N_T + N_{PI}))$.

3.3.2 *Time Complexity.* The analysis of time complexity is based on the following two assumptions. First, logic simulation of a test vector requires $O(N_G)$ basic operations. Secondly, fault simulation of a test vector for a single fault requires $O(N_G)$ basic operations.

The time complexity of the IFC algorithm is analyzed as follows. In Step 1, the cost of fault simulation without fault dropping is $O(N_F N_T N_G)$. In Step 2, the cost of finding essential faults is $O(N_F)$. Besides, the costs of extracting a single component and mapping it are $O(N_G)$ and $O(N_T N_{PI})$, respectively. Therefore, the overall complexity of Step 2 is $O(N_F(N_T N_{PI} + N_G))$. In Step 3, the cost of computing IFSs is $O(N_F^2 N_T^2)$. In Steps 4 and 5, the costs of sorting IFSs and test vectors detecting every fault are $O(N_F log_2 N_F)$ and $O(N_F N_T log_2 N_T)$, respectively. In Step 6, the complexity of component extraction is $O(N_F N_T N_G)$. This is because a component is extracted a number of times $O(N_T)$ if it is incompatible with existing compatibility sets. However, from our experimental results the average number of times a component is extracted for a fault is one. The complexity of mapping components of remaining faults to existing compatibility sets is $O(N_F N_T^2 N_{PI})$. Therefore, the overall complexity of Step 6 is $O(N_F N_T(N_T N_{PI} + N_G))$.

Based on our experimental analysis of the different phases of IFC (see Table XI), we noticed that most of the runtime of IFC is spent in computing the IFSs and matching the remaining faults. Hence, Steps 3 and 6 are the dominating sources of time consumption.

The time complexity of the CBC algorithm is computed as follows. In Step 1, the cost of fault simulation without fault dropping is $O(N_F N_T N_G)$. The cost of sorting test vectors in Step 2 is $O(N_T log_2 N_T)$. In Step 3, the cost of component generation is $O(N_F N_G)$. The cost of sorting test vectors in Step 4 is $O(N_T log_2 N_T)$. In Step 5, the cost of dropping redundant components using fault simulation with dropping is $O(N_F^2 N_G)$. The cost of merging components in Step 6 is $O(N_F)$. In Step 7, the cost of classifying test vectors is $O(N_F N_T N_{PI})$.

Computing the cost of processing class zero test vectors involves the following steps (see Figure 8). In Step 1, the cost of sorting class zero test vectors is $O(N_T log_2 N_T)$. In Step 2, the cost of computing blockage values for all class zero test vectors is $O(N_F^2 N_T N_{PI})$. In Step 3, the cost of moving components to appropriate test vectors is $O(N_F N_T)$ and the cost of updating S_{comp} for all components belonging to class zero test vectors is $O(N_F N_T N_{PI})$. Note that only components

whose S_{comp} contains eliminated and/or modified test vectors should be updated. The cost of updating blockage values for all class zero test vectors is $O(N_F^2 N_T^2 N_{PI})$. The cost of sorting test vectors in Step 4 is $O(N_T log_2 N_T)$. Finally, the cost of processing remaining class zero test vectors is $O(N_F N_T^2 N_{PI})$.

Based on our experimental analysis of the class zero algorithm (see Table XV), we noticed that most of the runtime of the algorithm is spent in computing class zero test vector blockage values and updating S_{comp} and blockage values of components. Hence, Steps 2, 3.3, and 3.4 are the dominating sources of time consumption.

After processing class zero test vectors, components of test vectors are merged and test vectors are reclassified. The costs of merging components and reclassifying test vectors are $O(N_F)$ and $O(N_F N_T N_{PI})$, respectively. After that, class one test vectors are processed as shown in Figure 9. In Step 1, the complexity of finding class one test vectors is $O(N_T)$. Besides, the complexity of computing S_{cand} of a CC belonging to class one test vector is $O(N_F N_{PI})$. This process costs $O(N_T N_F N_{PI})$ for all CCs. Thus, the complexity of Step 1 is $O(N_F N_T N_{PI})$. In Step 2, the cost of finding class one potential test vectors is $O(N_T)$. Furthermore, the cost of Step 2.1 is $O(N_T^3)$. The cost of sorting test vectors in Step 2.2 is $O(N_T^2 log_2 N_T)$. Therefore, the time complexity of Step 2 is $O(N_T^3)$. In Step 3, the cost of sorting class one potential test vectors is $O(N_T log_2 N_T)$. Computing the complexity of merging a class one potential test vector in Step 4 involves the following steps. First, the complexity of moving the CC to an appropriate test vector is $O(N_T)$. Secondly, the complexity of moving remaining components to appropriate test vectors is $O(N_F N_T)$. Thirdly, the complexity of reclassifying test vectors is $O(N_F N_T N_{PI})$. Therefore, the complexity of Step 4 is $O(N_F N_T^2 N_{PI})$. Hence, from the above analysis, it can be seen that the complexity of processing class one test vectors is $O(N_F N_T^2 N_{PI})$.

After processing class one test vectors, components of test vectors are merged and test vectors are reclassified. The costs of merging components and reclassifying test vectors are $O(N_F)$ and $O(N_F N_T N_{PI})$, respectively. After that, class i test vectors, where $i > 1$, are processed. As can be seen from the algorithm in Figure 11, the complexity of processing remaining classes is $O(N_F N_T^2 N_{PI})$.

Based on our experimental analysis of the different phases of CBC (see Table XV), we noticed that the CBC algorithm spends most of its runtime in the component generation, component elimination, and blockage value computation phases. Hence, Steps 3 and 5 in Figure 6 and Step 2 in Figure 8 are the dominating sources of time consumption.

4. EXPERIMENTAL RESULTS

In order to demonstrate the effectiveness of the IFC and CBC algorithms, we have performed experiments on a number of the ISCAS85 and full-scanned versions of ISCAS89 benchmark circuits. The experiments were run on a SUN Ultra60 (UltraSparc II-450 MHz) with a RAM of 512 MB. We have used test sets generated by HITEC [Niermann and Patel 1991]. In addition, we have used the fault simulator HOPE [Lee and Ha 1996] for fault simulation purposes and the test relaxation algorithm in [El-Maleh and Al-Suwaiyan 2002] for component generation.

Table IX summarizes the features of benchmark circuits we have used in our

Table IX. Benchmark circuits.

Cct	$\#\ Inputs$	$\#\ Outputs$	$\#\ Gates$	$\#\ TVs$	$\#\ CFs$	$\#\ DFs$	FC
$c2670$	233	140	1193	154	2747	2630	95.741
$c3540$	50	22	1669	350	3428	2895	84.452
$c5315$	178	123	2307	193	5350	5291	98.897
$s13207.1f$	700	790	7951	633	9815	9664	98.462
$s15850.1f$	611	684	9772	657	11725	11335	96.674
$s208.1f$	18	9	104	78	217	217	100
$s3271f$	142	130	1572	256	3270	3270	100
$s3330f$	172	205	1789	704	2870	2870	100
$s3384f$	226	209	1685	240	3380	3380	100
$s38417f$	1664	1742	22179	1472	31180	31004	99.436
$s38584f$	1464	1730	19253	1174	36303	34797	95.852
$s4863f$	153	120	2342	132	4764	4764	100
$s5378f$	214	228	2779	359	4603	4563	99.131
$s6669f$	322	294	3080	138	6684	6684	100
$s9234.1f$	247	250	5597	620	6927	6475	93.475

Table X. Results by the RM, GC, and IFC algorithms.

Cct	ROF	RM	GC			IFC	
	$\#\ TVs$	$\#\ TVs$	$\#\ Comp$	$\#\ TVs$	$Time\ (sec.)$ $Total$	$\#\ TVs$	$Time\ (sec.)$ $Total$
$c2670$	106	100	761	99	8.03	96	6.95
$c3540$	83	80	657	83	9.08	83	8.98
$c5315$	119	106	1491	117	34.95	103	31
$s13207.1f$	476	252	3516	248	339.93	243	169
$s15850.1f$	456	181	4135	169	463.95	144	249
$s208.1f$	33	33	94	33	0.009	32	0.93
$s3271f$	115	76	1212	69	14.97	61	7.02
$s3330f$	277	248	1263	233	11.01	208	9
$s3384f$	82	75	1048	73	15.01	72	7.97
$s38417f$	822	187	12215	173	5327.3	145	2072
$s38584f$	819	232	16086	210	9250	145	2590
$s4863f$	65	59	607	52	24.01	49	25.96
$s5378f$	252	145	1460	130	34.95	123	23
$s6669f$	52	42	1286	40	60.01	35	37.91
$s9234.1f$	375	202	2093	185	104.01	172	68.06

experiments. The first column gives the circuit name. Columns two through eight give the number of primary inputs, number of primary outputs, number of gates, number of Test Vectors (TVs), number of Collapsed Faults (CFs), number of Detected Faults (DFs), and Fault Coverage (FC), respectively.

In Table X, we report the results of applying the Random Merging (RM), Graph Coloring (GC), and IFC algorithms on the test sets after they are compacted by ROF. The first column gives the circuit name. The second and third columns give test set sizes after applying ROF and RM, respectively. Columns four through six give the results of the GC algorithm. The number of components obtained after dropping redundant ones is given under the column headed $\#Comp$. Test set sizes are given under the column headed $\#TVs$. The total time required by the GC algorithm is given under the column headed $Total$. Columns seven to eight give the results of the IFC algorithm. Test set sizes are given under the column headed

Table XI. Analysis of the different phases of IFC.

Cct	# *EFs*	# *CSs After Mat. EFs*	# *IFSs*	*Size of Max. IFS*	*Avg. # TVs*		*Max. # TVs Tried Per Fault*	*Time (sec.)*			
					Per Fault	*Tried Per Fault*		*FS ND*	*Mat. EFs*	*Build. IFSs*	*Mat. Rem. Faults*
*c*2670	269	91	787	26	22.2	1	2	0.95	0.06	1.96	3.96
*c*3540	343	74	665	22	13.62	1.1	3	0.95	0.07	0.97	7
*c*5315	302	85	1206	19	18.6	1.2	10	1.93	1.03	4	24.1
*s*13207.1*f*	1153	208	2906	63	106.8	1.03	12	15	10.06	61.97	75
*s*15850.1*f*	982	125	3301	62	95.61	1.04	16	23	15	56.05	151
*s*208.1*f*	59	32	64	7	7.48	1.01	2	0.005	0.9	0.003	0.01
*s*3271*f*	170	49	985	19	22.3	1.07	8	0.05	0.93	1.06	5
*s*3330*f*	358	198	730	37	51.6	1.04	4	1.04	0.96	3	3.94
*s*3384*f*	135	70	1221	10	25.93	1	4	0.95	0.02	1.97	5
*s*38417*f*	1913	106	9466	107	176	1.1	51	87	74	753	1133
*s*38584*f*	1670	126	10405	129	189.23	1.02	51	89.04	66	1212	1189
*s*4863*f*	227	36	1473	14	15.16	1.16	6	0.96	0.98	2.05	21.95
*s*5378*f*	320	109	1341	37	55.32	1.02	9	2.04	1.01	5.96	14
*s*6669*f*	100	23	2340	12	15.94	1.1	6	0.96	0.95	8.02	27.04
*s*9234.1*f*	722	157	1439	66	61.33	1.03	18	9	5.06	11	42.02

Table XII. Results by the Iter_IFC algorithm.

Cct	*IFC*	*Iter_IFC*		
	# *TVs*	# *TVs*	# *Iterations*	*Time (sec.)*
*c*2670	96	85	6	42.07
*c*3540	83	75	3	26.95
*c*5315	103	86	4	88.04
*s*13207.1*f*	243	238	2	473.12
*s*15850.1*f*	144	129	1	374.95
*s*208.1*f*	32	32	1	0.01
*s*3271*f*	61	60	2	18.98
*s*3330*f*	208	196	3	30.02
*s*3384*f*	72	72	1	7.07
*s*38417*f*	145	120	2	3775.06
*s*38584*f*	145	124	3	8217.08
*s*4863*f*	49	42	3	70.88
*s*5378*f*	123	117	6	109
*s*6669*f*	35	30	4	175.01
*s*9234.1*f*	172	155	4	200.93

$\#TVs$. The total time required by the IFC algorithm is given under the column headed $Total$.

The GC algorithm is called the Brelaz Color-Degree algorithm and is explained in [Mchugh 1990]. It proceeds as follows. First, an incompatibility graph is built. In this graph, nodes correspond to components and an edge exists between two nodes if their corresponding components are incompatible. Secondly, as long as the number of uncolored nodes is not zero, a node n^* is selected such that it has the maximum number of adjacent nodes. Now, n^* is colored with the current color c_k. Then, for every node n_i that is compatible with n^* and can be colored with c_k, it is colored with c_k. After that, the incompatibility graph is updated.

As can be seen from Table X, for most of the circuits, the GC algorithm is able

to compute test sets whose sizes are smaller than the sizes of the test sets obtained by RM. This observation reveals the potential of the TVD technique. Test sets computed by the GC algorithm are as much as 11.9% smaller than those computed by RM, e.g. 1% smaller for c2670, 9.5% smaller for s38584f, and 11.9% smaller for s4863f.

It can be seen that the results obtained by the IFC algorithm are better than those obtained by the RM and GC algorithms. The percentage improvement over the RM algorithm varies between 3% for s208.1f and 37.5% for s38584f. On the other hand, the percentage improvement over the GC algorithm varies between 1.4% for s3384f and 31% for s38584f. The runtime of the IFC algorithm is better than that of the GC algorithm.

In Table XI, we provide a detailed analysis of the IFC algorithm. The first column gives the circuit name. The second and third columns give the number of essential faults in the test set and the number of compatibility sets created after matching essential faults, respectively. The fourth and fifth columns give the number of independent fault sets and the maximum size of an independent fault set, respectively. The sixth column gives the average number of test vectors that detect a fault. The seventh and eighth columns give the average and maximum number of components generated per fault during the process of fault matching. Columns nine to twelve indicate the time taken by the different phases of the IFC algorithm. Column nine gives the time taken by fault simulation without dropping. Column ten gives the time taken for matching essential faults. Column eleven gives the time taken for building independent fault sets. Finally, column twelve gives the time taken for matching remaining faults.

The following observations can be made from the information in Table XI. First, an average of five essential faults are mapped to a compatibility set. Secondly, the average number of components generated for a fault is one. This indicates that on average, a component is mapped successfully to a compatibility set from the first trial. Thirdly, the most time consuming phases in the IFC algorithm are the phases of building the independent fault sets and the phase of matching the non-essential faults.

Our implementation of building the independent fault sets has a complexity of $O(N_F^2 N_T^2)$. However, a more efficient implementation can be achieved by finding pairwise independent faults and then solving a clique partitioning problem. Finding pairwise independent faults can be implemented efficiently using appropriate data structures. This will be investigated in future work.

The step of matching non-essential faults is time consuming mainly due to the generation of components. This step can be speeded up by reducing the number of components that need to be generated. This can be achieved by fault simulating the test vectors resulting from matching essential faults and dropping the detected non-essential faults. This will also be investigated in future work.

For large circuits with large number of faults, fault simulation without dropping can be also time consuming. The speed of fault simulation without dropping can be improved by employing the X-algorithm [Akers et al. 1990]. The X-algorithm, based on logic simulation and value justification, can significantly reduce the number of faults that need to be injected. Furthermore, double detection fault simulation

Table XIII. Results of applying CBC on test sets first compacted by ROF+RM.

Cct	*RM*	*CBC*			
	# TVs	*# TVs*			*Time (sec.)*
		After Class 0	*After Class 1*	*After Remaining Classes*	*Total*
*c*2670	100	94	94	94	10
*c*3540	80	78	78	78	13.02
*c*5315	106	96	96	95	29.03
*s*13207.1*f*	252	243	243	243	443
*s*15850.1*f*	181	147	145	145	476.02
*s*208.1*f*	33	33	33	33	0.01
*s*3271*f*	76	66	65	65	15.95
*s*3330*f*	248	226	223	223	27
*s*3384*f*	75	72	72	72	11.02
*s*38417*f*	187	146	143	143	5750
*s*38584*f*	232	159	153	153	8813
*s*4863*f*	59	52	52	52	24.04
*s*5378*f*	145	122	117	116	52
*s*6669*f*	42	37	37	37	50.1
*s*9234.1*f*	202	168	166	163	136

can be used to speed up fault simulation without dropping. The impact of double detection on the quality of the compacted test sets will be investigated in future work.

Critical Path Tracing (CPT) [Abramovici et al. 1984; Abramovici et al. 1990] can also be used to speed up fault simulation without dropping. CPT deals with faults implicitly. Therefore, fault simulation, fault collapsing, fault partitioning, fault insertion, and fault dropping are not needed. Furthermore, although CPT is an approximate method, it was experimentally shown in [Abramovici et al. 1984] that the impact of approximation is negligible. CPT can be implemented to be as fast as concurrent fault simulation [Abramovici et al. 1990].

In Table XII, we give the results of applying the iterative IFC algorithm on test sets first compacted by ROF. The first column gives the circuit name. The second column gives the test set sizes after running IFC for one iteration. The third column gives the test set sizes after applying IFC iteratively until no improvement is noticed. The fourth column gives the number of iterations that were run. Finally, the fifth column gives the time taken by the iterative IFC algorithm.

It can be seen that Iter_IFC improves over both RM and IFC. The percentage improvement over RM varies from 3% to 46.6%, e.g. 3% for s208.1f, 35.8% for s38417f, and 46.6% for s38584f. On the other hand, the percentage improvement over IFC varies from 1.6% to 17.2%, e.g. 1.6% for s3271f, 14.5% for s38584f, and 17.2% for s38417f.

In Table XIII, we give the results of applying the CBC algorithm on test sets first compacted by ROF+RM[1]. The unspecified entries in test vectors are randomly filled. The first column gives the circuit name. The second column gives the test set sizes after applying RM. Columns three to five give the test set sizes after processing

[1]ROF+RM is an abbreviation for ROF followed by RM.

Table XIV. Statistics about the test sets compacted by ROF+RM.

Cct	*# Elim. Comp.*	*# TVs*			*Maximun Degree of Hardness*
		Class 0	*Class* 1	*Class* 1 *Potential*	
*c*2670	1869	12	46	0	18
*c*3540	2263	0	12	0	23
*c*5315	3758	77	21	0	29
*s*13207.1*f*	6219	252	150	0	49
*s*15850.1*f*	7283	180	63	4	39
*s*208.1*f*	123	0	22	0	15
*s*3271*f*	2090	76	39	1	7
*s*3330*f*	1595	194	134	5	21
*s*3384*f*	2363	3	24	0	3
*s*38417*f*	19055	186	24	24	70
*s*38584*f*	20386	232	10	10	41
*s*4863*f*	4173	58	17	0	9
*s*5378*f*	3085	144	17	16	14
*s*6669*f*	5426	41	15	0	13
*s*9234.1*f*	4414	194	64	2	28

Table XV. Analysis of some phases of CBC.

Cct	*Time* (*sec.*)							
	FS ND	*Comp. Gen.*	*Comp. Elim.*	*TV Recons.*	*TV Classif.*	*TV Block.*	*Proc. Class* 0	*Proc. Class* 1
*c*2670	0.05	3	5.03	0.002	0.93	0.001	0.002	0.01
*c*3540	0.96	5.03	7.02	0.001	0.002	0	0	0.0001
*c*5315	1.01	7.05	16.94	0.004	0.04	1	1.96	0.004
*s*13207.1*f*	8.01	26.92	123.01	0.04	9	173.97	257	0.99
*s*15850.1*f*	9.01	73.93	177.07	0.92	3.09	115.92	200	6.91
*s*208.1*f*	0.001	0.003	0.002	0.00004	0.0002	0	0.00001	0.0001
*s*3271*f*	0.94	2.05	6.93	0.002	0.02	4	5.02	0.03
*s*3330*f*	1.02	2.03	5.92	0.003	0.06	7.03	14.01	2
*s*3384*f*	0.94	2.06	6.97	0.003	0.03	0.0005	0.001	0.01
*s*38417*f*	20	588	1102	72.06	34.95	2310	2919	119
*s*38584*f*	25.04	730.06	1369.02	79	87	4029	5340	256
*s*4863*f*	0.06	10.92	12	0.001	0.008	0.1	1.03	0.0003
*s*5378*f*	1.92	5	19.03	0.004	0.96	10.04	19.03	2.05
*s*6669*f*	0.06	18.03	25.96	0.006	0.03	4.93	6	0.002
*s*9234.1*f*	4	12	54.02	0.007	1.93	24.02	48.03	2.01

test vectors belonging to class zero, class one, and remaining classes, respectively. The runtime of the CBC algorithm is given under the column headed *Total*.

In Table XIV, we give some statistics about the test sets compacted by ROF+RM. The statistics are generated while the CBC algorithm processes the test sets. The second column gives the number of eliminated components dropped by fault simulation. The third, fourth, and fifth columns indicate the size of class 0, size of class 1 after processing class 0, number of class one potential test vectors, respectively. The maximum degree of hardness computed before processing remaining classes is given under the column headed *Maximun Degree of Hardness*.

The following observations can be made from the results in Table XIII and information in Table XIV. For circuits c3540 and s208.1f, the size of class zero is zero. This indicates that every test vector has at least one CC. However, for the

circuit c3540, although it does not have class zero test vectors, some improvement is noticed after processing class zero test vectors. This is because some test vectors are eliminated in the component generation phase since they do not detect essential faults. Another interesting observation is that not all class zero test vectors can be eliminated. This is because while processing class zero test vectors, the S_{comp} of some components will become empty which makes their parent test vectors become non-class zero test vectors. In addition, S_{comp} may contain only class zero test vectors.

It is also observed that although the size of class one is large, the number of class one potential test vectors is very small. In fact, the number of class one potential test vectors is zero for most of the circuits. In general, if the size of class i, where $i > 0$, is greater than zero and the number of class i potential test vectors is zero, this indicates that every class i test vector has at least one CC whose S_{cand} is empty. It should also be observed that not all potential test vectors can be eliminated. This is because potential test vectors can be damaged. A potential test vector is said to be damaged if the S_{cand} of one or more of its CCs become empty. In addition, a potential test vector is damaged if one or more of its components become CCs and/or if it receives one or more CCs from other potential test vectors.

As can be seen from the results in Table XIII, the CBC algorithm reduces the test sets by as much as 34%, e.g. 2.5% for c3540, 23.5% for s38417f, and 34% for s38584f. It should be observed that the improvements achieved after processing class one are very small. This is due to the reasons explained above.

In Table XV, we give a detailed analysis of some phases of the CBC algorithm. The first column gives the circuit name. Column two gives the time taken by fault simulation without dropping. Columns three and four give the time taken for generating components and dropping redundant ones, respectively. Columns five and six give the time taken for reconstructing and reclassifying test vectors, respectively. Column seven gives the time taken for computing the initial blockage values for all class zero test vectors (see Step 2 in Figure 8). It should be pointed out that the computation of test vector blockage is part of the phase of processing class zero test vectors. Finally, columns eight and nine give the time taken for processing class zero and class one test vectors, respectively.

As can be seen from the table, most of the runtime of the CBC algorithm is spent in the component generation, component elimination, and blockage value computation phases. Component generation can be speeded up by first generating the components for essential faults and fault simulating them to drop all detected non-essential faults. Hence, the number of components that need to be generated for remaining faults will be reduced. Furthermore, the time requirement of the component elimination phase is reduced since less components are generated. Other techniques for speeding up the component elimination phase will be investigated in future work.

Another interesting observation that can be seen from the table is that our current implementation of the blockage value computation phase is time consuming. More efficient techniques for computing test vector blockage and other heuristics will be investigated in future work.

Table XVI shows the results of applying the CBC algorithm on test sets com-

Table XVI. Results of applying CBC on test sets compacted by ROF+IFC and ROF+Iter_IFC.

Cct	$ROF+IFC$	$ROF+IFC+CBC$		$ROF+Iter\_IFC$	$ROF+Iter\_IFC+CBC$	
	$\#TVs$	$\#TVs$	$Time\ (sec.)$	$\#TVs$	$\#TVs$	$Time\ (sec.)$
$c2670$	96	93	10	85	83	9.97
$c3540$	83	78	13.95	75	73	13
$c5315$	103	78	31.96	86	70	33.02
$s13207.1f$	243	239	174.02	238	236	180.93
$s15850.1f$	144	134	248.01	129	126	250.05
$s208.1f$	32	32	0.01	32	32	0.01
$s3271f$	61	59	9.98	60	58	10
$s3330f$	208	201	11.93	196	188	7.01
$s3384f$	72	72	11.04	72	72	11.96
$s38417f$	145	122	4684	120	111	3433
$s38584f$	145	129	5322	124	121	3563
$s4863f$	49	44	24.01	42	42	25.96
$s5378f$	123	113	37.96	117	112	26.97
$s6669f$	35	35	43	30	30	44.03
$s9234.1f$	172	156	82	155	143	72

pacted by ROF+IFC and ROF+Iter_IFC. The unspecified entries in test vectors are randomly filled. The first column gives the circuit name. The second and third columns give the test set sizes for ROF+IFC and ROF+IFC+CBC, respectively. The fourth column gives the runtime of applying CBC on test sets compacted by ROF+IFC. The fifth and sixth columns give the test set sizes for ROF+Iter_IFC and ROF+Iter_IFC+CBC, respectively. The last column gives the runtime of applying CBC on test sets compacted by ROF+Iter_IFC.

From Table XVI, it can be seen that the CBC algorithm reduces the test sets compacted by ROF+IFC by as much as 23.5%, e.g. 3% for c2670, 16% for s38417f, and 23.5% for c5315. It can also be seen that the CBC algorithm reduces the test sets compacted by ROF+Iter_IFC by as much as 18.6%, e.g. 2.7% for c3540, 7.7% for s9234.1f, and 18.6 c5315.

The improvement of ROF+IFC+CBC over ROF+RM+CBC varies from 1.1% to 18%. ROF+Iter_IFC+CBC improves over both ROF+RM+CBC and ROF+IFC+CBC. The improvement over ROF+RM+CBC varies from 3% to 26.3%. On the other hand, the improvement over ROF+IFC+CBC varies from 1.3% to 14.3%. Therefore, we propose a hybrid static compaction algorithm that is composed of ROF, Iter_IFC, and CBC. Given a test set, by first running ROF, redundant test vectors are dropped quickly. Then, Iter_IFC is used to compact the test set. Finally, CBC is used to optimize the size of the test set as much as possible.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have introduced the concept of Test Vector Decomposition (TVD). Using TVD, a test vector can be decomposed into its atomic components. Then, it is eliminated if its components can be all moved to other test vectors. Based on this approach, we have proposed two new static compaction algorithms. They are Independent Fault Clustering (IFC) and Class-Based Clustering (CBC). In IFC, independent faults are found and then compatible faults are matched together. Two independent faults can be mapped to the same compatibility set if their components

are compatible. On the other hand, in CBC, classes of test vectors are formed and then test vectors are processed in increasing order of their degree of hardness. At the end, every test vector represents a cluster whose components originally belong to test vectors in different classes. Experimental results are reported to demonstrate the effectiveness of the two algorithms. In general, the IFC algorithm has achieved an improvement of as much as 37.5% over random merging. Besides, the iterative version of IFC has achieved an improvement of as much as 46.6% over random merging and 17.2% over IFC. Furthermore, the CBC algorithm has achieved a test set reduction of as much as 34%.

In the future, we will investigate the impact of the critical path tracing and double detection algorithms on the quality of compacted test sets. Besides, we will consider reducing the complexity of non-essential fault matching in the IFC algorithm by fault simulating test vectors resulting from essential fault matching and then dropping the detected non-essential faults. Furthermore, we will consider improving the current implementation of building the independent fault sets. Finally, we will consider improving the time consuming phases in CBC.

ACKNOWLEDGMENT

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

REFERENCES

Abramovici, M., Bruer, M. A., and Friedman, A. D. 1990. *Digital Systems Testing and Testable Design*. IEEE, Piscataway, NJ.

Abramovici, M., Menon, P. R., and Miller, D. T. 1984. Critical Path Tracing – An Alternative to Fault Simulation. *IEEE Design and Test*, 83–92.

Akers, S. B. and Joseph, C. 1987. On the Role of Independent Fault Sets in the Generation of Minimal Test Sets. In *Proc. of the International Test Conference*. IEEE, Washington, D.C., USA, 1100–1107.

Akers, S. B., Krishamurthy, B., Park, S., and Swaminathan, A. 1990. Why Is Less Information From Logic Simulation More Useful in Fault Simulation? In *Proc. of the International Test Conference*. IEEE, 786–800.

Akers, S. B. and Krishnamurthy, B. 1989. Test Counting: A Tool for VLSI Testing. *IEEE Design and Test of Computers 6,* 5 (Oct.), 58–73.

Ayari, B. and Kaminska, B. 1994. A New Dynamic Test Vector Compaction for Automatic Test Pattern Generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 13,* 3 (March), 353–358.

Boateng, K. O., Konishi, H., and Nakata, T. 2001. A Method of Static Compaction of Test Stimuli. In *Proc. of the Asian Test Symposium*. IEEE, Kyoto, Japan, 137–142.

Chang, J.-S. and Lin, C.-S. 1995. Test Set Compaction for Combinational Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 14,* 11 (Nov.), 1370–1378.

El-Maleh, A. and Al-Suwaiyan, A. 2002. An Efficient Test Relaxation Technique for Combinational and Full-Scan Sequential Circuits. In *Proc. of the VLSI Test Symposium*. IEEE, Monterey, CA, 53–59.

Flores, P. F., Neto, H. C., and Marques-Silva, J. P. 1999. On Applying Set Covering Models to Test Set Compaction. In *Proc. of the Ninth Great Lakes Symposium on VLSI*. IEEE, Ypsilanti, MI, USA, 8–11.

Garey, M. R. and Johnson, D. S. 1979. *Computers and Intractability: A Guid to the Theory of NP-Completeness*. W.H. Freedman, San Francisco.

Hamzaoglu, I. and Patel, J. H. 1998. Test Set Compaction Algorithms for Combinational Circuits. In *Proc. of the International Conference on Computer-Aided Design*. IEEE, San Jose, CA, USA, 283–289.

Hamzaoglu, I. and Patel, J. H. 2000. Test Set Compaction Algorithms for Combinational Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 19,* 8 (Aug.), 957–963.

Hochbaum, D. S. 1996. An Optimal Test Compression Procedure for Combinational Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 15,* 10 (Oct.), 1294–1299.

Kajihara, S. and Miyase, K. 2001. On Identifying Don't Care Inputs of Test Patterns for Combinational Circuits. In *IEEE/ACM Int'l Conference on Computer-Aided Design*. IEEE, San Jose, CA, USA, 364–369.

Kajihara, S., Pomeranz, I., Kinoshita, K., and Reddy, S. M. 1994. On Compacting Test Sets by Addition and Removal of Test Vectors. In *VLSI Test Symposium*. IEEE, Cherry Hill, NJ, USA, 25–28.

Kajihara, S., Pomeranz, I., Kinoshita, K., and Reddy, S. M. 1995. Cost Effective Generation of Minimal Test Sets for Stuck-At Faults in Combinational Logic Circuits. *IEEE Transactions on Computer-Aided Design 14,* 12 (Dec.), 1496–1504.

Krishnamurthy, B. and Akers, S. B. 1984. On the Complexity of Estimating the Size of a Test Set. *IEEE Transactions on Computers C-33,* 8 (Aug.), 750–753.

Lee, H. K. and Ha, D. S. 1996. HOPE: An Efficient Parallel Fault Simulator for Synchronous Sequential Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 15,* 9 (Sept.), 1048–1058.

Lin, X., Rajski, J., Pomeranz, I., and Reddy, S. M. 2001. On Static Test Compaction and Test Pattern Ordering for Scan Designs. In *Proc. of the Int'l Test Conference*. IEEE, Baltimore, MD, USA, 1088–1098.

Mchugh, J. 1990. *Algorithmic Graph Theory*. Prentice Hall, NJ.

Miyase, K., Kajihara, S., and Reddy, S. M. 2002. A Method of Static Test Compaction Based on Don't Care Identification. In *Proc. of the First IEEE Int'l Workshop on Electronic Design, Test, and Application*. IEEE, Christchurch, New Zealand, 392–395.

Niermann, T. M. and Patel, J. H. 1991. HITEC: A Test Generation Package for Sequential Circuits. In *Proc. of the European Conference on Design Automation*. IEEE, Amsterdam, Netherlands, 214–218.

Pomeranz, I., Reddy, L. N., and Reddy, S. M. 1993. Compacttest: A Method to Generate Compact Test Sets for Combinational Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 12,* 7 (July), 1040–1049.

Pomeranz, I. and Reddy, S. M. 1992. Generalization of Independent Faults for Transition Faults. In *Proc. of the VLSI Test Symposium*. IEEE, Atlantic City, NJ, USA, 7–12.

Pomeranz, I. and Reddy, S. M. 2001. Forward-Looking Fault Simulation for Improved Static Compaction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 20,* 10 (Oct.), 1262–1265.

Reddy, L. N., Pomeranz, I., and Reddy, S. M. 1992. ROTCO: A Reverse Order Test Compaction Technique. In *Proc. of the EURO-ASIC Conference*. IEEE, Paris , France, 189–194.

Schulz, M. H., Trischler, E., and Sarfert, T. M. 1988. SOCRATES: A Highly Efficient Automatic Test Pattern Generation System. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 7,* 1 (Jan.), 126–137.

Tromp, G.-J. 1991. Minimal Test Sets for Combinational Circuits. In *Proc. of the International Test Conference*. IEEE, 204–209.

Wang, J. C. and Stabler, E. P. 1995. Collective Test Generation and Test Set Compaction. In *Proc. of the International Symposium on Circuits and Systems*. IEEE, Seattle, WA, USA, 2008–2011.