A Class-based Clustering Static Compaction Technique for Combinational Circuits

Aiman El-Maleh and Yahya Osais

The 16th International Conference on Microelectronics, pp. 522–525, 6-8 Dec. 2004.

A CLASS-BASED CLUSTERING STATIC COMPACTION TECHNIQUE FOR COMBINATIONAL CIRCUITS

Aiman H. El-Maleh and Yahya E. Osais

King Fahd University of Petroleum & Minerals, Computer Engineering Department, Dhahran 31261, Saudi Arabia
{aimane,yosais}@ccse.kfupm.edu.sa

ABSTRACT

Static compaction based on test vector merging is a very simple and efficient technique. However, for a highly incompatible test set, merging achieves little reduction. In this paper, we propose a new static compaction technique in which a test vector is 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.

1. INTRODUCTION

As chip sizes increase, the volume of test data rises sharply. This results in a corresponding increase in test application time and tester storage space. Compression and compaction algorithms are used to reduce test data volume. In compression, 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. In compaction, however, test data is manipulated to drop unnecessary data, provided that there is no drop in fault coverage.

Test compaction algorithms are either dynamic or static. Dynamic compaction algorithms are integrated into the test generation process. On the other hand, static compaction algorithms are applied on test sets after they are generated.

According to [1], static compaction algorithms for combinational circuits can be classified into three broad categories: (1) Redundant Vector Elimination, (2) Test Vector Modification, and (3) Test Vector Addition and Removal. In the first category, the size of a test set is reduced by removing redundant test vectors. A redundant test vector detects no essential faults. An essential fault is a fault detected only by one test vector. Redundant test vectors can be identified using a set cover, e.g. [2], or fault simulation, e.g. [3]. In the second category, however, test vectors are modified such that they can be merged together, e.g. [4], pruned, e.g. [5], or decomposed, e.g. [6]. Finally, in the third category, new test vectors are added to the given test set to remove some of the already existing test vectors, e.g. [7].

In this paper, we propose a new static compaction algorithm for combinational circuits. The algorithm is referred to as Class-Based Clustering (CBC) and is based on Test Vector Decomposition (TVD). Algorithms based on TVD are considered an improvement over merging-based static compaction algorithms.

The paper is structured as follows. First, we start with some preliminaries necessary to understand our CBC algorithm. Then, the CBC algorithm is described. After that, we discuss the experimental results. Finally, we conclude by summarizing the results of the paper and their significance.

2. PRELIMINARIES

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 [8], 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.

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 nth 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 is 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 iff $S_j \neq \phi$ 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 $|\alpha|$ iff for every CC c_i in α, c_i is movable.

3. ALGORITHM DESCRIPTION

The CBC algorithm is shown as Algorithm 1 and proceeds as follows. First, the given test set is fault simulated without fault dropping. This step is performed to find the number and set of test vectors that detect every fault. Second, 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. Components are generated 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 Algorithm 2. First, test vectors are 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 need 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 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.

Algorithm 1 CBC(T)

1. Fault simulate T without fault dropping.
2. Sort test vectors in increasing order of their number of faults.
3. Generate atomic components.
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 Algorithm 2).
9. For every test vector, merge its components together.
10. Reclassify test vectors.
11. Process class one test vectors (see Algorithm 3).
12. For every test vector, merge its components together.
13. Reclassify test vectors.
14. Process class i test vectors, where $i > 1$ (see Algorithm 5).

Algorithm 2 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.

Algorithm 3 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 Algorithm 4). 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.

S_{comp} of every component must be updated after eliminating every test vector.

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 Algorithm 3. 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

Algorithm 4 Merge_Class_1_Potential_TVs

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.

Algorithm 5 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.

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 Algorithm 4). 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.

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 Algorithm 5. 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.

4. EXPERIMENTAL RESULTS

In order to demonstrate the effectiveness of the CBC algorithm, 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 [9]. In addition, we have used the fault simulator HOPE [10] for fault simulation purposes and the test relaxation algorithm in [8] for component generation.

Table 1. Results of applying CBC on test sets first compacted by ROF+RM.

Cct	TS	ROF	RM	CBC			
	# TVs	# TVs	# TVs	# TVs			Time (sec.)
				After Class 0	After Class 1	After Remaining Classes	Total
c2670	154	106	100	94	94	94	10
c3540	350	83	80	78	78	78	13.02
c5315	193	119	106	96	96	95	29.03
s13207.1f	633	476	252	243	243	243	443
s15850.1f	657	456	181	147	145	145	476.02
s208.1f	78	33	33	33	33	33	0.01
s3271f	256	115	76	66	65	65	15.95
s3330f	704	277	248	226	223	223	27
s3384f	240	82	75	72	72	72	11.02
s38417f	1472	822	187	146	143	143	5750
s38584f	1174	819	232	159	153	153	8813
s4863f	132	65	59	52	52	52	24.04
s5378f	359	252	145	122	117	116	52
s6669f	138	52	42	37	37	37	50.1
s9234.1f	620	375	202	168	166	163	136

In Table 1, 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 and second columns give the circuit names and original test set sizes, respectively. The third and fourth columns give the test set sizes after applying ROF and RM, respectively. Columns five to eight give the test set sizes after processing test vectors belonging to class zero, class one, and remaining classes, respectively. The runtime of the CBC algorithm is given under the column headed $Total$.

As can be seen from the results, 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 i, where $i > 0$, are very small.

5. CONCLUSIONS

In this paper, we have proposed an efficient static compaction technique for combinational circuits. The technique is based on decomposing test vectors into their atomic components and classifying them into classes that are processed to cluster the components in such a way that results in a more compacted test set. Based on experimental results, the proposed technique has achieved a test set reduction of as much as 34% when applied on test sets initially compacted by ROF+RM.

6. ACKNOWLEDGMENT

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

7. REFERENCES

[1] Yahya E. Osais, "Efficient Static Compaction Algorithms for Combinational Circuits Based on Test Relaxation," *MS Thesis, King Fahd University of Petroleum and Minerals*, Oct. 2003.

[2] Kwame Osei Boateng, Hideaki Konishi, and Tsuneo Nakata, "A Method of Static Compaction of Test Stimuli," in *Proc. of the Asian Test Symposium*, Nov. 2001, pp. 137–142.

[3] Irith Pomeranz and Sudhakar M. Reddy, "Forward-Looking Fault Simulation for Improved Static Compaction," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 10, pp. 1262–1265, Oct. 2001.

[4] Jau-Shien Chang and Chen-Shang Lin, "Test Set Compaction for Combinational Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 11, pp. 1370–1378, Nov. 1995.

[5] Lakshmi N. Reddy, Irith Pomeranz, and Sudhakar M. Reddy, "ROTCO: A Reverse Order Test Compaction Technique," in *Proc. of the EURO-ASIC Conference*, June 1992, pp. 189–194.

[6] Yahya E. Osais and Aiman H. El-Maleh, "A static test compaction technique for combinational circuits based on independent fault clustering," in *Proc. of the 10th IEEE Int'l Conference on Electronics, Circuits, and Systems*, Dec. 2003 (To Appear).

[7] Seiji Kajihara, Irith Pomranz, Kozo Kinoshita, and Sudhakar M. Reddy, "On Compacting Test Sets by Addition and Removal of Test Vectors," in *VLSI Test Symposium*, April 1994, pp. 25–28.

[8] Aiman El-Maleh and Ali Al-Suwaiyan, "An Efficient Test Relaxation Technique for Combinational and Full-Scan Sequential Circuits," in *Proc. of the VLSI Test Symposium*, 2002, pp. 53–59.

[9] T. M. Niermann and J. H. Patel, "HITEC: A Test Generation Package for Sequential Circuits," in *Proc. of the European Conference on Design Automation*, Feb. 1991, pp. 214–218.

[10] Hyung Ki Lee and Dong Sam Ha, "HOPE: An Efficient Parallel Fault Simulator for Synchronous Sequential Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 9, pp. 1048–1058, Sept. 1996.

[1] ROF+RM is an abbreviation for Reverse-Order Fault simulation followed by Random Merging.