PAPER *Special Issue on Test and Verification of VLSI*

Test Generation for Test Compression Based on Statistical Coding

Hideyuki ICHIHARA[†], ***Regular Member***, **Atsuhiro OGAWA**[†*], ***Nonmember***, **Tomoo INOUE**[†], ***and*** **Akio TAMURA**[†], ***Regular Members***

SUMMARY Test compression/decompression is an efficient method for reducing the test application cost. In this paper we propose a test generation method for obtaining test-patterns suitable to test compression by statistical coding. In general, an ATPG generates a test-pattern that includes don't-care values. In our method, such don't-care values are specified based on an estimation of the final probability of 0/1 occurrence in the resultant test set. Experimental results show that our method can generate test patterns that are able to be highly compressed by statistical coding, in small computational time.
key words: *VLSI test, test compression, statistical code, test generation, automatic test equipment*

1. Introduction

As the size and complexity of VLSI circuits increase, the size of test sets for such circuits also increases. The increase in test set size requires larger storage and longer time to transport test sets from the storage device of a VLSI tester (ATE) to the circuit-under-test (CUT). Especially in core based designs, the amount of test sets tends to increase because each individual core requires a separate test set.

To alleviate this problem, two kinds of useful approaches are proposed: test compaction [1]–[3] and test compression [4]–[8]. The former, test compaction is a method for reducing the number of test-patterns while the fault coverage achieved by the test-patterns is kept high. The latter, test compression is a method to encode a given test set before test application, and then decode it by a decoder on a target CUT. We focus on test compression in this paper.

Figure 1 shows a scheme underlying test compression methods. In this scheme, a given test set is compressed by a data compression technique and stored in a VLSI tester storage. While a CUT on a chip is tested, the compressed test set is transported to a decompressor on the chip, and then decompressed and fed to the CUT. The compressed test set can achieve the reduction of the time for test transportation, not just the size of the test storage device. Time required for decompression is negligible because the clock frequency in a chip is faster than that of a VLSI tester.

A test compression/decompression method using the Burrows-Wheeler transformation and run-length code for test compression has been proposed [4]. This method can achieve high compression of test sets, so that the transportation time from the test storage device to CUTs decreases greatly. However, the decompression is complex, and hence was implemented in software.

The methods proposed in [5] and [6] are based on cyclical scan registers. In these methods, a given test set/sequence is converted to a difference vector sequence, and it is compressed by run-length code [5] or Golomb code [6]. Although the methods can accomplish a high compression of test sets, additional circuits for configuring the cyclical scan registers and controlling a testing clock are needed.

The methods of [7] and [8] employ *statistical coding* as a compression method for a test set. Since a statistical code decides the length of a code word according to the probability of occurrences of each unique pattern, the encoding compresses a test set efficiently. In addition, the decoder (decompressor) can be implemented with a simple finite state machine (FSM). In [7], a BIST architecture for non-scan sequential circuits based on a statistical code has been presented. This method is useful for circuits in which the number of primary inputs is small. The paper [8] presented a statistical code called a selective code based on the Huffman code, and proposed a method for compressing scan vectors by means of the code. Although the average length of a selective code word is a little longer than that of a Huffman code word, the decoder for the selective code needs fewer states than that for the Huffman code.

In compression methods based on statistical coding, the compression ratio of a test set depends on the

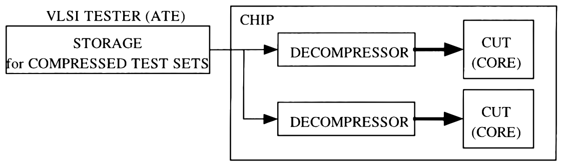


Fig. 1 Scheme of test compression.

Manuscript received January 22, 2002.
Manuscript revised May 29, 2002.
[†]The authors are with the Faculty of Information Sciences, Hiroshima City University, Hiroshima-shi, 731–3194 Japan.
[*]Presently, with AS System Business Division, ADVANTEST Corporation.

probability of occurrence of each pattern in the test set. The authors [9] proposed a method to transform generated test sets so that the compression ratio of the test set is enhanced by varying the probability of occurrence of each pattern. In the method, fault simulation is repeated to keep fault coverage achieved by the test set. Repeating fault simulation makes the computational time large, but the method of [9] drastically improves the compression of test sets.

In general, an ATPG (test-pattern generator) generates a test-pattern that include don't-care values for a fault. Such don't-care values of the generated test-pattern are specified by a certain method (e.g., randomly), and then fault simulation for the completely-specified test-patterns is performed. If we can specify don't-care values in each generated test-pattern so that it is suitable for statistical coding, the resultant test set can be compressed efficiently.

In this paper, we propose a test generation method for compression by statistical coding. In this method, a test-pattern that has don't-cares is specified by estimating the probability of occurrence of each pattern. The probability estimation in the proposed method is updated along with the progress of the test generation process, and the code word of each pattern is determined step by step, i.e., the final code word is determined *dynamically*. In contrast, note that we can consider that the method [9], mentioned above, is *static* because the transformation is performed after a completely-specified test set is generated.

The remainder of this paper is organized as follows. Section 2 reviews a statistical encoding method of a test set. Section 3 describes the motivation and the strategy of our method, and then proposes the procedure of our method. Section 4 gives experimental results, and Sect. 5 concludes the paper.

2. Test Compression by Statistical Code

Here, we illustrate the method proposed in [8] as an example of test compression using statistical encoding. To encode a given test set, the test set is partitioned into n-bit blocks, i.e., each block is an n-bit pattern. Table 1 shows a test set which consists of twelve test-patterns divided into 4-bit blocks. The reason for partitioning a test-pattern into blocks is to keep the complexity of a decompression circuit and decompression delays low. Each block pattern is mapped into a variable-length code word. The length of a code word depends on the probability that each pattern appears in the test set. The more frequently a pattern occurs, the shorter the length of a code word for it. Table 2 shows the number n_i and the probability p_i of occurrence of each unique pattern x_i in the test set of Table 1. In addition, Huffman code words, as an example of statistical encoding, are shown, too.

The average length of a statistical code is derived from $\sum_{i=1}^{m} p_i \cdot w_i$, where w_i is the length of the code word corresponding to pattern x_i. As shown in Table 2, the average length of a Huffman code word in this example is 3.3125. It is well-known that a lower bound of the average length of a code word used for encoding an information source can be expressed by its entropy. The entropy of a test set is given by $H = -\sum_{i=1}^{m} p_i \cdot \log_2 p_i$, where the test set includes m unique patterns $x_1, x_2, \ldots, x_m$ with probabilities of occurrence $p_1, p_2, \ldots, p_m$, respectively. The average length of a Huffman code word for a block pattern is the closest to the entropy of a test set. The entropy of the test set of Table 1 is 3.2740, it is close to the average length of a Huffman code word 3.3125.

An important property of a statistical code is prefix-free, i.e., a code word is never a prefix of another code word. Note that Huffman code is prefix-free. If the decoder for a prefix-free statistical code is constructed as a finite state machine (FSM), the decoder requires m states, where m is the number of unique patterns. For example, the decoder for the Huffman code in Table 2 requires fifteen states.

Table 1 Test set divided into 4-bit blocks.

test 1:	1111 0101 0011 1111 1011 1110 1101 1011
test 2:	1111 1111 1111 1111 0000 0000 0000 0000
test 3:	1001 0010 0110 0111 1110 1101 0110 1110
test 4:	1111 1111 1110 0110 1111 1001 1111 1011
test 5:	1111 1111 0111 **1010** 1111 1111 0111 1111
test 6:	0010 1110 1000 0100 1111 1111 1111 1111
test 7:	0110 1011 1011 1011 1101 0111 1011 0111
test 8:	1100 1000 1010 0111 0101 1011 1111 1101
test 9:	1011 0100 1101 1101 1110 1111 1111 1111
test10:	1111 1011 0111 0101 1111 0111 1111 1101
test11:	1100 1101 1001 1110 1110 1101 1011 0110
test12:	0111 0010 1111 1111 0111 1111 1011 1111

Table 2 Block pattern probability table and Huffman code words.

i	x_i	n_i	p_i	Huffman Code
1	1111	30	0.3125	11
2	1011	12	0.1250	100
3	0111	10	0.1042	010
4	1101	9	0.0938	000
5	1110	8	0.0833	1011
6	0110	5	0.0521	0011
7	0000	4	0.0417	10101
8	0010	3	0.0313	01111
9	1001	3	0.0313	01110
10	0101	3	0.0313	01101
11	1100	2	0.0208	01100
12	0100	2	0.0208	00101
13	1000	2	0.0208	00100
14	1010	2	0.0208	101001
15	0011	1	0.0104	101000
entropy: 3.2740				ave. 3.3125
#states of FSM				15

3. Highly-Compressible Test Generation by Statistical Coding

3.1 Motivation

As mentioned in the previous section, the entropy of a test set gives a lower bound of the average length of a statistical code word for the encoded test set. Accordingly, in general, a statistical encoding method makes the average length of its code word close to the entropy of a target test set. Consequently, in test compression using a statistical coding, we can expect that, if we can reduce the entropy of a test set, the average length of a code word for the test set becomes small. Note that, although the example of a statistical code in the previous section is Huffman code, this observation will be applicable not only to Huffman encoding but also to a general statistical coding, e.g., comma coding [7], selective coding [8].

The following Lemma 1, which has been proved in [9], leads a method to generate a test set with low entropy. Thus, we apply Lemma 1 to test pattern generation.

Lemma 1: Consider a block with pattern x_a in a test set partitioned into fixed-length blocks. If pattern x_a of the block can be replaced with pattern x_b that occurs more frequently than x_a, i.e., $p_a < p_b$, then the entropy of the encoded test set decreases. □

Consider the example in Table 1 again. Assume that pattern "1010" of a block (which is boldfaced) is replaced with pattern "1111," which occurs most frequently. In this case, the number of block pattern "1111" increases by 1 to 31 and the number of block pattern "1010" decreases by 1 to 1, so that the entropy becomes 3.2284, which is smaller than the previous entropy 3.3125.

3.2 Strategy

Prior to the specification of our test generation method, first we describe an underlying test generation algorithm. For a fault selected from a given fault list, a test-pattern that includes don't-care values is generated. The don't-care values of the generated test-pattern are specified by a certain procedure (e.g., at random straightforwardly). Fault simulation for the completely-specified test-pattern is performed and the detectable faults identified by the fault simulation are dropped from the fault list. These processes are repeated until the fault list becomes empty.

Our strategy is to specify don't-care values so that the resultant test set is suitable to compression by statistical coding. That is, after generating a test-pattern that includes don't-care values in the above test generation, we specify the don't-care values so that the entropy of the resultant test set becomes low. In the concrete, based on Lemma 1, we specify a block pattern including don't-care values so that the block pattern becomes the pattern that occurs most frequently in the test set. However, the probabilities of occurrence of block patterns can not be obtained until the completely-specified test set is generated. Therefore, we use a method for estimating the occurrence probability, which is called *block probability estimation.*

In order to precisely estimate the probability of occurrence of a block pattern, we introduce two parameters N_X and N_R which are concerned with the number of block patterns.

3.3 Procedure for Generating Compressible Test

Here, we present a straightforward procedure for compressible test generation. Figures 2 and 3 show procedures of our method. Procedure Generate_Compressible_Test in Fig. 2 gives a overview of the proposed method. It receives four parameters F, n, N_X

```
1  Generate_Compressible_Test($F, n, N_X, N_R$)
2    $F$: fault list;
3    $n$: length of a block;
4    $N_X, N_R$: number of X blocks;
5  {
6    Set test set $T = \phi$;
7    Set block pattern probability table $BPPT = \phi$;
8    for(each fault $f$ in $F$){
9      Generate test $t$ for $f$ and add $t$ into $T$;
10     Divide test set $T$ into $n$-bit blocks;
11     if(the number of X blocks in $T$ is over $N_X$)
12       Specify_X_Block($T, BPPT, N_R$);
13     Perform fault simulation with tests completely specified in $T$, and drop the detected faults from $F$;
14   }
15   Specify__X_Block($T, BPPT, 0$);
16   return $T$;
17 }
```

Fig. 2 Compressible test generation procedure.

```
1  Specify_X_Block($T, BPPT, N_R$)
2  {
3    Count the number of each unique block pattern in $T$, and update $BPPT$;
4    for(each test $t_i \in T$ in the order of generation){
5      for(each block $b_j$ in $t_i$){
6        if($b_j$ is not X block) continue;
7        for(each $b_k \in BPPT$ in descending order of the probability of block pattern){
8          if($b_j$ is compatible with $b_k$){
9            Replace $b_j$ with $b_k$;
10           Update $BPPT$;
11           break;
12         }
13       }
14     }
15     if(the number of X blocks in $T$ is equal to $N_R$)
16       return;
17   }
18 }
```

Fig. 3 X block specification procedure.

and N_R. F is a given fault list, and n is the length of block pattern. N_X and N_R are the numbers of *X blocks* in which patterns include don't-care values. In the procedure, T is a test set to be generated and $BPPT$ is a block pattern probability table which shows the probability of each pattern like Table 2, and it is used for block probability estimation.

Basically, for each fault f in fault list F, our procedure generates test-pattern t which may include X blocks (lines 8–9 in Fig. 2). After dividing generated test set T into n-bit blocks (line 10), if the number of X blocks is over N_X, the procedure calls sub-procedure Specify_X_Blocks that specifies X blocks (lines 11–12). If there newly exist completely-specified tests, fault simulation is performed for the tests (line 13). When fault list F is empty, the don't-care values remaining in test set T are specified (line 15) and then the procedure returns test set T (line 16).

Procedure Specify_X_Block is shown in Fig. 3. The procedure receives three parameters T, $BPPT$ and N_R. In the procedure, $t_i \in T$ denotes the i-th generated test pattern, and $p_i \in BPPT$ is a block pattern whose frequency of occurrence is the i-th of all block patterns in $BPPT$. After being called from procedure Generate_Compressible_Test, the procedure counts the number of each unique block pattern in T and then updates block pattern probability table $BPPT$ (line 3 in Fig. 3). For each test t_i in the order of generation, it is checked if each X block b_j in test t_i is compatible with block b_k appearing more frequently (lines 4–8). Note that X block b_j is said to be compatible with completely-specified block b_k if the don't care values of b_j can be specified so as to b_j is equal to b_k. If X block b_j is compatible with block b_k(line 8), the procedure replaces X block b_j with block b_k and updates $BPPT$ (lines 9–10). If the number of X blocks in test set T is equal to N_R (line 15), the process returns to procedure Generate_Compressible_Test.

We demonstrate our procedure by Figs. 4 and 5. Suppose that procedure Generate_Compressible_Test is called with $n = 4$, $N_X = 7$ and $N_R = 3$, and then it generates four test patterns without calling procedure Specify_X_Block. The partitioned test set is shown in Fig. 4 (a). Note that, in this example, a block can lie on more than two consecutive test-patterns of a test set, e.g., a block "XXX1" on test-patterns t_1 and t_2 in Fig. 4 (a). The corresponding decompressor can be implemented without extra schemes (see [8] in detail). After generating four test patterns, since the number of X blocks in the test set is over N_X (in this case, they are 8 and 7, respectively) Specify_X_Block is called. In procedure Specify_X_Block, the block pattern probability table shown in Fig. 5 is constructed. Note that the elements of table is sorted in descending order of the frequency of each block pattern, and an X block (e.g., "X00X") is counted as all of the compatible block patterns (e.g., "0000," "0001," "1000" and "1001"). Based

	0	1	2	3	4	5	6	7	8	9
t1	0	1	0	1	0	X	0	1	X	X
t2	X	1	X	0	0	X	0	X	X	1
t3	X	1	0	X	X	1	X	1	0	0
t4	0	X	1	1	0	1	0	X	0	1

(a) Initial partitioned test set

	0	1	2	3	4	5	6	7	8	9
t1	0	1	0	1	0	1	0	1	0	1
t2	0	1	0	0	0	1	0	1	0	1
t3	0	1	0	1	X	1	X	1	0	0
t4	0	X	1	1	0	1	0	X	0	1

(b) After X block specification ($N_R = 3$)

Fig. 4 Example of test generation.

block pattern	#freq.
0101	7
0001	6
1101	4
...	...

Fig. 5 Block pattern probability table.

on the block pattern probability table, X blocks are specified until the number of X blocks remaining in the test set is equal to N_R (3 in this case). Figure 4 (b) shows the test set after the X block specification. Note that X blocks are specified in order of their generation (in this case, "0X01" in $t1$, "XXX1" in $t1$ and $t2$, "X00X" in $t2$, ...). Four X block patterns "0X01," "XXX1," "0XX1" and "X10X" are replaced with compatible pattern "0101" which occurs most frequently, and an X block pattern "X00X" is replaced with "0001" which occurs secondly frequently. In this case two test-patterns are completely specified, and hence fault simulation is performed for the two test-patterns in procedure Generate_Compressible_Test.

Here, we consider the effect of parameters N_X and N_R on the reduction of test set entropy. When large N_X is set, the number of test-patterns that are generated before specifying X blocks in procedure Specify_X_Block becomes large. Accordingly, the block probability estimation may be precise because it can be formed with many test-patterns. On the other hand, when large N_R is set, the number of X blocks that are specified by a call of Specify_X_Block becomes small, that is, X blocks are specified more cautiously. Consequently, large N_X and N_R are desirable for reducing the entropy of the generated test set.

From the viewpoint of the number of generated test-patterns, however, large N_X and N_R may increase the resultant number of test-patterns. The reasons are as follows. When large N_X is set, the number of test-patterns that are generated before specifying X blocks in procedure Specify_X_Block becomes large. Since fault simulation is performed only for completely-specified test-patterns, the number of times of fault simulation decreases. Accordingly, the number of faults dropped from a fault list by fault simulation also decreases, so that the number of faults targeted for test generation at line 9 in Fig. 2 increases. The increase of the target faults is directly linked to the increase of test-patterns. When large N_R is set, on the other hand, many incompletely-specified test-patterns remain in test set T after specifying X blocks. As a result, in the same way as large N_X, large N_R leads to the decrease of the number of times of fault simulation, so that it increases the number of test-patterns.

From the above discussion, there exist optimum N_X and N_R which achieve small test-patterns in number and the small entropy of the resultant test set. We consider optimal pair of N_X and N_R through the experiments shown in the next section.

Since the order of X block specification affects the resultant code word length, there may exist several methods for specifying X blocks other than the proposed X block specification in Fig. 3. However, we use Specify_X_Block, which specifies X blocks in order of their generation, because of the following several advantages. One is that the extra effort for the X block specification is small because a certain procedure for deciding an X block specification order is not required. Another reason is that the proposed method is applicable to sequential circuits because specifying test-patterns in the order of their generation suites to a requirement that fault simulation for a sequential circuit should be performed in the order of a test sequence.

4. Experimental Results

We implemented the proposed method in C and applied it to ISCAS benchmark circuits on a workstation Sun Ultra 10 (UltraSPARC-IIi, 440 MHz). The implementation is based on test generation algorithm SOCRATES [10]. To examine the effect of N_X and N_R, we attempted our method with pairs of N_X={10, 50, 100, 300, 500, 700, 1000, 1200, 1500, 2000} and N_R={1, 10, 20, 50, 100, 200, 500, 1000, 1500}. When N_R is larger than N_X, such pair is neglected. We also tried some different block sizes (4, 6, 8) used to partition a test set. Here, we show only the case where the block size is 8 because the results for block size 4 and 6 are similar to those for block size 8. The statistical coding used is that described in Sect. 2. Note that our method can be applicable to different statistical codings, e.g., comma coding [7] and selective coding [8].

In Table 3 we report the results of our method. After the circuit name, the number of primary inputs is shown. In the next five columns, we show the results of test sets generated without our method, i.e., don't-care values of each test-pattern are specified randomly and fault simulation is performed for every generated test-pattern. Columns tests, comp., ratio and states denote the number of test-patterns, the total length (bits) of the test set encoded by Huffman coding, the ratio of the total length of a test set after Huffman encoding to that before, and the number of states in the finite state machine needed to decode the encoded test sets, respectively. Column time denotes the computational time required for procedure Generate_Compressible_Test (in Fig. 2), which consists of deterministic test generation, fault simulation and X block specification. The next seven columns show the results of our test generation. The results shown in Table 3 are the smallest total lengths of test sets compressed of all results for different pairs of N_X and N_R. The last two columns show the values of N_X and N_R, respectively, when the smallest compressed test sets are derived.

The boldface values in the ninth column mean that the total lengths of compressed test sets generated by our method are smaller than those of the test sets generated without our method. As you can see, our method can reduce the total length of compressed test set for almost circuits. Especially for c2670, c5315, s9234, s13207, s15850 and s38417, the total lengths of test sets generated by our method can be reduced to one third of those without our method. In addition, comparing the numbers of states in the decoders for both methods (shown in the sixth column and the eleventh column in Table 3), we can see that the proposed method can

Table 3 Results of compressible test generation.

		without our method					with our method						
circuit	inputs	tests	comp.	ratio	states	time	tests	comp.	ratio	states	time	N_X	N_R
c432	36	63	2028	0.89	162	0.08	78	**1088**	0.38	34	0.1	10	1
c499	41	65	2400	0.90	176	0.13	81	**2115**	0.63	102	0.16	10	1
c880	60	74	4238	0.95	222	0.09	107	**2124**	0.33	50	0.22	50	1
c1355	41	90	3327	0.90	202	1.31	134	3653	0.66	131	1.73	10	1
c1908	33	152	4798	0.95	230	0.46	194	5286	0.82	171	0.53	10	1
c2670	233	196	38495	0.84	255	1.03	199	**10200**	0.21	66	1.55	50	1
c3540	50	197	9649	0.97	252	1.11	273	**5931**	0.43	74	1.95	10	1
c5315	178	189	33570	0.99	255	3.96	283	**11076**	0.21	83	6.17	50	10
c6288	32	42	1126	0.83	111	2.35	130	2510	0.60	75	2.01	10	1
c7552	207	264	54357	0.99	255	50.07	420	**22935**	0.26	135	22.89	100	10
s9234	247	526	127822	0.98	255	35.67	672	**33445**	0.20	102	28.63	50	10
s13207	700	667	452026	0.96	255	21.74	1084	**108101**	0.14	137	30.95	300	1
s15850	611	527	317759	0.98	255	22.89	835	**79154**	0.15	152	30.86	10	1
s35932	1763	79	139271	0.99	255	33.76	574	**129385**	0.12	17	83.26	100	10
s38417	1664	1721	2644136	0.92	255	115.41	3868	**881040**	0.13	118	409.17	100	50
s38584	1464	824	1205819	0.99	255	322.52	4737	**916349**	0.13	94	760.5	100	1

reduce the number of states in the corresponding decoders. This is because our method reduces the number of unique block patterns according to Lemma 1. Note that, as mentioned in Sect. 2, the number of states in a decoders for a test set is equal to the number of unique block patterns in the test set.

The computational time for the proposed method, shown in the twelfth column in Table 3, is comparable to that for the test generation without our method, shown in the seventh column. However, for relatively large circuits (i.e., s35932, s38417 and s38584), our computational times become twice as large as the original ones. This is due to ad hoc implementation of the proposed algorithm — the first objective of this experiment is to confirm the effectiveness of our method in reducing the size of the compressed test set, and hence the program was not efficient for the circuit size. Therefore, we consider that if we tune up the program, especially the sorting procedure that works for updating block pattern probability table (at lines 3 and 10 in Fig. 3), the computational time for large circuits can be reduced.

In order to clarify the effectiveness of our method, we show the entropy and the average length of a Huffman code word for the test sets in Table 4. The second and third columns show the entropy of the test sets generated with and without our method, respectively. The forth column shows the ratio of the entropy with our method to that without our method. The fifth, sixth and seventh columns show the average length of a Huffman code word of two test sets and its ratio. The average length of a code word is the product of the compression ratio shown in Table 3 and the block size (8 in our experiment). The average values of every columns appear in the last row of Table 4. From this table, we can see that the entropy of test sets generated with our method becomes, on average, a third as small as that without our method. Moreover, the average length of a Huffman code word also decreases by a third. This is because Huffman coding is an optimal statistical coding in which the average length of the code word is the closest to the entropy of a target data source. Hence, our method is suitable to test compression by statistical coding, especially Huffman coding. As a result, our method can effectively reduce the entropy of test sets.

To clarify the effect of estimation parameters N_X and N_R, in Fig. 6, we illustrate the total length of compressed test set for c7552 according to the parameters. As you can see, an optimal total length of compressed test set is achieved when N_X and N_R are appropriately small ($N_X = 100$ and $N_R = 10$). The tendency that appropriately small N_X and N_R can produce optimal test sets is seen in the results for other circuits, as you can also see in Table 3. Thus, in the case where the proposed method is used in practical, it may be able to generate a highly compressible test set with appropriately small N_X and N_R.

The total length of compressed test set is given by the product of the compression ratio, which is the ratio of the total length of test sets after Huffman coding to that before Huffman coding, and the number of test-patterns. Figures 7 and 8 show the compres-

Table 4 Entropy and average length of Huffman code word.

	entropy			av. Huffman code word		
circuit	w/o	with	ratio	w/o	with	ratio
c432	7.12	3.04	0.43	7.15	3.10	0.43
c499	7.13	5.04	0.71	7.2	5.09	0.71
c880	7.61	2.62	0.34	7.64	2.65	0.35
c1355	7.18	5.29	0.74	7.21	5.32	0.74
c1908	7.61	6.57	0.86	7.65	6.61	0.86
c2670	6.7	1.56	0.23	6.74	1.76	0.26
c3540	7.8	3.46	0.44	7.84	3.48	0.44
c5315	7.96	1.48	0.19	7.98	1.76	0.22
c6288	6.63	4.8	0.72	6.7	4.83	0.72
c7552	7.93	1.94	0.24	7.96	2.11	0.27
s9234	7.84	1.25	0.16	7.87	1.61	0.2
s13207	7.71	0.35	0.05	7.75	1.14	0.15
s15850	7.87	0.55	0.07	7.89	1.24	0.16
s35932	7.99	0.13	0.02	8.00	1.02	0.13
s38417	7.36	0.27	0.04	7.39	1.10	0.15
s38584	7.99	0.19	0.02	8.00	1.06	0.13
average	7.53	2.41	0.32	7.56	2.74	0.36

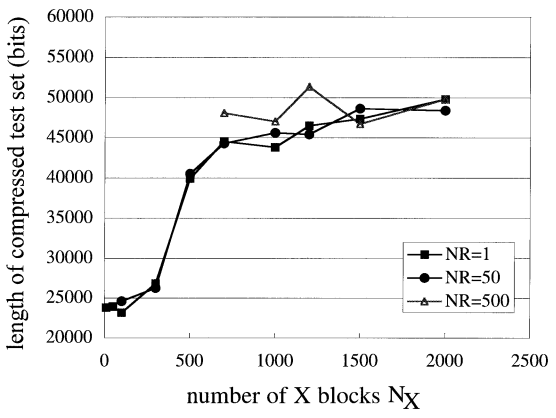


Fig. 6 Total length of compressed test sets for c7552.

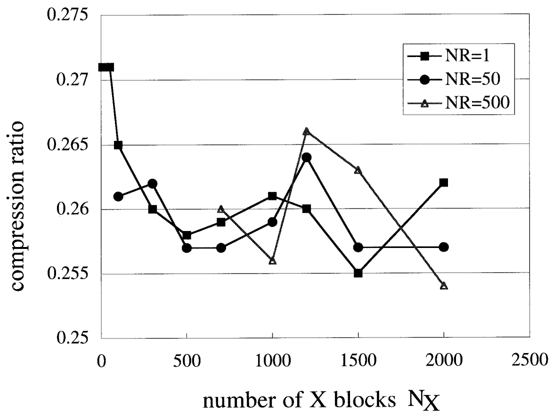


Fig. 7 Compression ratio for c7552.

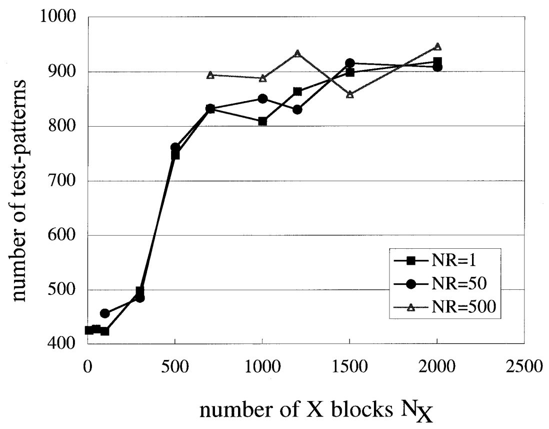


Fig. 8 Number of test-patterns for c7552.

Table 5 Results of static compression [9].

circuit	comp.	time[sec.]
c432	666	1.84
c499	1567	13.81
c880	1082	4.52
c1355	2579	84.4
c1908	2859	181
c2670	8011	170.8
c3540	2821	115.1
c5315	6178	188.7
c6288	307	54.58

sion ratio and the number of test-patterns generated for c7552, respectively, in the same manner as Fig. 6. From Fig. 7, the compression ratio decreases with N_X increasing, i.e., large N_X can reduce the entropy of a test set. From Fig. 8, in contrast, the number of test-patterns increases with N_X increasing. Although these tendencies derived from two figures were discussed in Sect. 3.3, comparing Fig. 6 to Figs. 7 and 8, the transition of the total length of compressed test set, shown in Fig. 6, is dominated by that of the number of test-patterns, shown in Fig. 8, rather than that of compression ratio. Thus, if a much smaller compressed test set is desired, it is required to generate small test-patterns in number without increasing the entropy of the test set.

Table 5 shows the results of static compression in [9]. The blocks size used to partition a test set is eight, which is equal to the results of Table 3. The results for the larger circuits than c6288 can not be obtained in our experimental environment because of a large computational time and a shortage of working memory required. Comparing the results of Table 5 to Table 3, although the total lengths of test sets generated by our method is longer than that by static compression, the computational time by our method is considerably smaller than that by static compression. In addition, our method can obtain the results for the circuits whose results are not obtained by the method [9].

5. Conclusions

This paper proposed a method for generating a test set suitable for compression by statistical encoding. Experimental results show that our method can reduce the length of compressed test set with small extra effort. From the analysis of the experimental results, we can see that the estimation parameters N_X and N_R should be appropriately small for practical use.

In order to reduce the size of a compressed test set further, it is important to generate small test-patterns in number. As an approach to this, we are considering the combination of our test generation and test compaction, i.e., generating a small test set while reducing its entropy. Some preliminary results show the effectiveness in reducing the resultant size of compressed test sets [11]. As a future work, we pursue this study further.

References

[1] S. Kajihara, I. Pomeranz, K. Kinoshita, and S.M. Reddy, "Cost-effective generation of minimal test sets for stuck-at faults in combinational logic circuits," Trans. CAD, vol.14, no.12, pp.1496–1503, 1995.

[2] I. Pomeranz and S.M. Reddy, "On test compaction objectives for combinational and sequential circuits," Proc. VLSI Design Conference., pp.279–294, Jan. 1998.

[3] I. Hamzaoglu and J.H. Patel, "Test set compaction algorithm for combinational circuits," Proc. International Conference on Computer-Aided Design, pp.283–289, 1998.

[4] M. Ishida, D.S. Ha, and T. Yamaguchi, "COMPACT: A hybrid method for compressing test data," Proc. VLSI Test Symposium, pp.62–69, 1998.

[5] A. Jas and N.A. Touba, "Test vector decompression via cyclical scan chains and its application to testing core-based designs," Proc. International Test Conference, pp.458–464, 1998.

[6] A. Chandra and K. Chakrabarty, "Test data compression for system-on-a-chip using Golomb codes," Proc. VLSI Test Symposium, pp.113–120, 2000.

[7] V. Iyengar, K. Chakrabarty, and B. T Murray, "Built-in self testing of sequential circuits using precomputed test sets," Proc. VLSI Test Symposium, pp.418–423, 1998.

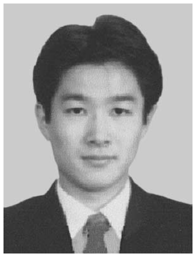
[8] A. Jas, J. Ghosh-Dastidar, and N.A. Touba, "Scan vector compression/decompression using statistical coding," Proc. VLSI Test Symposium, pp.114–120, 1999.

[9] H. Ichihara, K Kinoshita, I. Pomeranz, and S.M. Reddy, "Test transformation to improve compression by statistical encoding," Proc. VLSI Design, pp.294–299, 2000.

[10] M.H. Schulz, E. Trischler, and T.M. Sarfert, "SOCRATES: A highly efficient automatic test pattern generation system," IEEE Trans. Comput.-Aided Des. Integrated Circuits & Syst., vol.CAD-7, no.1, pp.126–137, 1988.

[11] H. Ichihara and T. Inoue, "Generating small test sets for test compression/decompression using statistical coding," Proc. First International Workshop on Electronic Design, Test & Applications, pp.396–400, 2002.

Hideyuki Ichihara received his M.E. and Ph. D degrees in Applied Physics from Osaka University, Japan, in 1997, 1999, respectively. He was a research scholar of University of Iowa, U.S.A. from February 1999 to July 1999. Since December 1999, he has been an assistant professor of Hiroshima City University in Japan. His research interests are test generation for logic circuits, logic optimization using test generation techniques and design for testability. He is a member of the IEEE.



Atsuhiro Ogawa received his Bachelor of Information Engineering degree from Hiroshima City University, Japan, in 2001. He is currently with ADVANTEST corporation.

Tomoo Inoue is an associate professor at the Faculty of Information Sciences, Hiroshima City University. His research interests include test generation, high-level synthesis and design for testability, and testing for field-programmable gate arrays. He received the BE, ME and PhD degrees from Meiji University, Kawasaki, Japan, in 1998, 1990 and 1997, respectively. Previously, he was an assistant professor at the Graduate School of Information Science, Nara Institute of Science and Technology. He is a member of the IEEE and IPSJ.

Akio Tamura received his B.S. degree in 1955 and Dr. Eng. degree in1997 from Kyushu University respectively. 1955 to 1987, he joined at Totsuka Works, Hitachi Ltd. and 1987 to 1994 he was a board director, senior chief engineer at Hitachi Microsoftware Systems Inc. Since 1994, he has been a professor at Faculty of Information Sciences, Hiroshima City University. His research interest includes Computer Graphics, CAD, CADG and electronic circuit design. He is a member of the IPSJ and IIEEJ.