A Geometric-Primitives-Based Compression Scheme for Testing Systems-on-a-Chip

Aiman El-Maleh[1], Saif al Zahir[2], and Esam Khan[1]

[1] King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia
[2] University of British Columbia, ECE Dept., Vancouver, B.C., Canada
Email: {aimane, esamkhan}@ccse.kfupm.edu.sa, saif_zahir@yahoo.com

Abstract

The increasing complexity of systems-on-a-chip with the accompanied increase in their test data size has made the need for test data reduction imperative. In this paper, we introduce a novel and very efficient lossless compression technique for testing systems-on-a-chip based on geometric shapes. The technique exploits reordering of test vectors to minimize the number of shapes needed to encode the test data. The effectiveness of the technique in achieving high compression ratio is demonstrated on the largest ISCAS85 and full-scanned versions of ISCAS89 benchmark circuits. In this paper, it is assumed that an embedded core will be used to execute the decompression algorithm and decompress the test data.

1. Introduction

With today's technology, it is possible to build complete systems containing millions of transistors on a single chip. Systems-on-a-chip (SOC) are comprised of a collection of pre-designed and pre-verified cores and user defined logic (UDL). As the complexity of systems-on-a-chip continues to increase, the difficulty and cost of testing such chips is increasing rapidly [11], [12]. To test a certain chip, the entire set of test vectors, for all the cores and components inside the chip, has to be stored in the tester memory. Then, during testing, the test data must be transferred to the chip under test and test responses collected from the chip to the tester as illustrated in Figure 1.

One of the challenges in testing SOC is dealing with the large size of test data that must be stored in the tester and transferred between the tester and the chip. The amount of time required to test a chip depends on the size of test data that has to be transferred from the tester to the chip and the channel capacity.

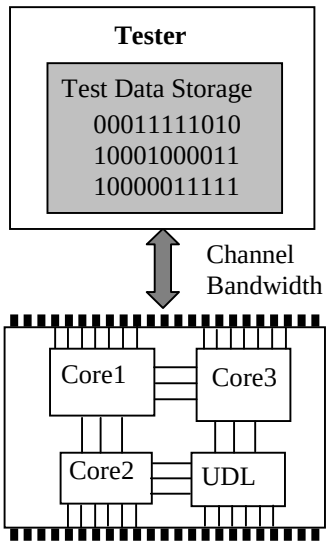


Figure 1. Test data transfer between the tester and the circuit under test.

The cost of automatic test equipment (ATE) increases significantly with the increase in their speed, channel capacity, and memory. As testers have limited speed, channel bandwidth, and memory, the need for test data reduction becomes imperative. To achieve such reduction, several compaction and lossless compression schemes were proposed in the literature.

The objective of test set compaction is to generate the minimum number of test vectors that achieve the desired fault coverage. There are two main types of compaction, static compaction and dynamic compaction. In static compaction, the number of test vectors is reduced after they have been generated. Examples of static compaction algorithms include reverse order fault simulation [15], forced pair merging [16], N_by_M [18], and redundant vector elimination (RVE) [14]. In dynamic compaction, the number of vectors is

minimized during the automatic test pattern generation (ATPG) process. Examples of dynamic compaction algorithms include COMPACTEST [17], and bottleneck removal [6].

In test data compression, the objective is to reduce the number of bits needed to represent the test data. For test data compression, it is essential that the compression is lossless. Run length coding, Huffman codes, Lempel-Ziv algorithms, and arithmetic codes are examples of lossless compression [13].

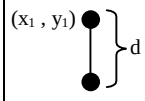
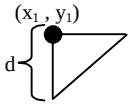
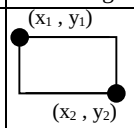
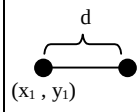
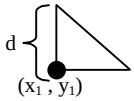
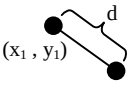
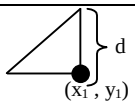
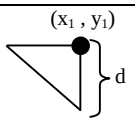
Several test data compression/decompression techniques were proposed in the literature. These techniques can be classifies into two categories; one is based on BIST and Pseudo-Random Generators (PRG) and the other is based on deterministic compression.

Examples of BIST-based compression techniques are test width compression [2], variable length reseeding [5], and Design For High Test Compression (DFHTC) [10].

Deterministic compression techniques take advantage of the high correlation between test vectors. One of these techniques is proposed in [1] and uses Burrows-wheeler (BW) transformation and a modified version of run-length coding to encode the test data. This technique has been improved in [3] by applying the GZIP compression scheme to strings that are not effectively compressed by run-length coding. Another technique proposed in [8] uses what is called variable-to-block run-length coding. In this technique, a codeword is used to encode a block of data based on the number of zeros followed by a one in that block. This technique is used for compressing fully-specified test data that feeds a cyclical scan chain. A cyclical scan chain is used to decompress this data and transfer it to the "test scan chain". Golomb code is a variable-to-variable run-length code that is used in [4] to enhance the scheme described above. It divides the runs into groups, each is of size *m*. The number of groups is determined by the length of the longest run, and the group size *m* is dependent on the distribution of test data. In [9], statistical coding is used for encoding deterministic test data. The technique uses a modified version of Huffman coding as to minimize the bits needed for codewords. Although this technique has less compression ratio than Huffman coding, the hardware implementation of the decoder is simpler. Another technique was proposed in [7] which performs decompression of test data based on an embedded processor. The technique is based on storing the differing bits between two test vectors. It divides each test vector into blocks and stores those blocks that are different from the preceding vector.

In this paper, we introduce a novel and very efficient compression scheme for deterministic testing of SOCs based on geometric shapes. This scheme is designed based on test cubes to maximize the compression ratio. Test vector decompression is performed on chip and is implemented either in hardware or software. For hardware decompression option, a decoding circuitry is placed on the chip to perform the decompression algorithm. However, for software decompression option, an embedded core is used to execute the decompression algorithm and decompress the test data, which is then applied to the circuit under test. The decompression algorithm can be stored in a ROM on the chip.

Table 1. The used primitive geometric shapes**.**

	Lines	Triangles	Rectangle
Type1			
Type 2			**X**
Type3			**X**
Type 4			**X**

2. The Proposed Encoding Algorithm

The proposed encoding algorithm is based on encoding the 0's or the 1's in a test set by geometric shapes. In this work, we limited those primitive shapes to the basic four, namely: point, line, triangle, and rectangle as shown in Table 1. These shapes are the most frequently encountered shapes in any test set. For the rectangles, two points are needed to encode the shape and each point costs $2*\log_2 N$, where N is the block dimension. However, lines and triangles can be represented by a point and a distance *d* and this reduces the number of bits needed to encode them by $(\log_2 N)-2$ in comparison to encoding them by two points. Two bits are used to determine the type of line or the type of triangle encoded.

Figure 2 shows the algorithm of the encoder, which consists of the following main steps:

(i) Test Set Sorting

Sorting the vectors in a test set is crucial and has a significant impact on the compression ratio. In this step, we aim at generating clusters of either 0's or 1's in such a way that it may partially or totally be fitted in one or more of the geometric shapes shown in Table 1. Several sorting scenarios have been considered and investigated. In this work, we used a simple correlation-based sorting technique. The sorting may be with respect to 0's (*0-*

```
Encoder (N)
    Sort_Test_Set ();
    Partition_Test_Set (N);
    For i = 1 to # of segments
       For j = 1 to # of blocks in i
          Extract_Shapes (1, j);
          α1 = Encode_Shapes ();
          Extract_Shapes (0, j);
          α0 = Encode_Shapes ();
          B = # of bits in j + 2;
          E = min (α0, α1,B);
          Store_Encoded_Bits ();
          E_total += E;
End Encoder;

Extract_Shapes(b, j)
  For each bit x in block j {
    If x = b Then {
      Find the largest line of each type started at x
      Find the largest triangle of each type such that
        x is the vertix of the right angle
      Find the largest rectangle such tha x is its up-
        left corner
    }
  }
  Solve a covering problem to find the best group of
    shapes covering all bits b in block j.
End Extract_Shapes;
```

Figure 2. Test vectors encoding algorithm.

sorting), to 1's (*1-sorting*) or to both 0's and 1's (*0/1-sorting*). The technique is based on finding the distance D between two vectors A and B that maximizes the clusters of 0's and 1's.

The distance D may be computed with respect to 0's (*0-distance*), to 1's (*1-distance*) or to 0's and 1's (*0/1-*distance) as follows:

$$D = \sum_{i=0}^{k-1} W(A_i, B_{i-1}) + W(A_i, B_i) + W(A_i, B_{i+1})$$

where k is the test vector length and $W(A_i, B_i)$ is the weight between bits A_i and B_i. Table 2, Table 3 and Table 4 specify the weights used in computing the 0-distance, the 1-distance, and the 0/1-distance between two vectors, respectively. Note that for i = 0, $W(A_i, B_{i-1}) = 0$ and for i = k-1, $W(A_i, B_{i+1}) = 0$.

The assignment of a 0.25 weight for an 'x' to each of its immediate neighbors be it an 'x' or the sorted bit ('0' for 0-sorting, '1' for 1-sorting and '0' and '1' for 0/1-sorting) is chosen due to the following reasons. First, this weight may help in completing, integrating, or generating additional geometric shapes that can lead to a better solution. Second, this can help in generating blocks filled by 'x's which can be minimally encoded. Different weights have been experimented with, and a weight of 0.25 has been found to produce better results in most of the cases.

Table 2. Weights for the 0-distance between two test vectors.

	0	1	x
0	1.0	0.0	0.25
1	0.0	0.0	0.0
x	0.25	0.0	0.25

Table 3. Weights for the 1-distance between two test vectors.

	0	1	x
0	0.0	0.0	0.0
1	0.0	1.0	0.25
x	0.0	0.25	0.25

Table 4. Weights for the 0/1-distance between two test vectors.

	0	1	x
0	1.0	0.0	0.25
1	0.0	1.0	0.25
x	0.25	0.25	0.25

Table 5. An example of test vector sorting.

Original Vectors	*v1*	0	0	1	X	1	0	X	X
	v2	0	X	1	1	0	0	0	1
	v3	1	1	X	1	1	X	0	1
Sorted Vectors (0-dist.)	*v2*	0	X	1	1	0	0	0	1
	v1	0	0	1	X	1	0	X	X
	v3	1	1	X	1	1	X	0	1
Sorted Vectors (1-dist.)	*v3*	1	1	X	1	1	X	0	1
	v2	0	X	1	1	0	0	0	1
	v1	0	0	1	X	1	0	X	X

In Table 5, we show a simple example to illustrate the impact of sorting on test vector compression. As can be seen, sorting the vectors based on the 0-distance requires the encoding of two triangles to encode the 0's. However, sorting the vectors based on the 1-distance requires the encoding of one triangle and two lines to encode the 1's. Thus, for this example sorting based on the 0-distance results in higher compression.

(ii) Test Set Partitioning

A set of sorted test vectors, M, is represented in a matrix form, RxC, where R is the number of test vectors and C is the length of each test vector. The test set is segmented into LxK blocks each of which is NxN bits, where L is equal to $\lceil R/N \rceil$ and K is equal to $\lceil C/N \rceil$. A *segment* consists of K blocks. In other words, the test set

is segmented into L segments each contains K blocks. For test vectors whose columns and/or rows are not divisible by the predetermined block dimension N, a partial block will be produced at the right end columns and/or the bottom rows of the test data. Since the size of such partial blocks can be deduced based on the number of vectors, the vector length, and the block dimension, the number of bits used to encode the coordinates of the geometric shapes can be less than $log_2 N$. The decoder recognizes those special cases and decodes them properly.

(iii) Encoding process

As mentioned earlier, the encoding process will be applied on each block independently. The procedure *Extract_Shapes(b)* will find the best group of shapes that cover the bits that are equal to *b* as shown in the algorithm. *Encode_Shapes* determines the number of bits, α, needed to encode this group of shapes. There are two cases that may occur:

(a) The block contains only 0's and x's or 1's and x's. In this case, the block can be encoded as a rectangle. However, instead of this it is encoded by the code 01 followed by the bit that fills the block. Hence, the number of bits to encode the block α = 3.

(b) The block needs to be encoded by a number of shapes. In this case, we need the following:

- 2 bits to indicate the existence of shapes and the type of bit encoded. If the encoded bit is 0, then the code is 10, otherwise it is 11.
- P = (2*Log_2 N – 3) bits to encode the number of shapes, S. If the number of shapes exceeds 2^P, then the number of bits needed to encode the shapes is certainly greater than the total number of bits in the block. In this case, the block is not encoded and the real data is stored.
- $\sum_{i=1}^{S} L_i$; where L_i is computed as follows
  - If shape i is a point, $L_i = 2 + 2*\log_2 N$ (shape type, coordinates).
  - If shape i is a line or a triangle, $L_i = 4 + 3*\log_2 N$ (shape type, type of line or triangle, point and distance)
  - If shape i is a rectangle, $L_i = 2 + 4*\log_2 N$ (shape type, 2 points)

Therefore, $\alpha = 2 + P + \sum_{i=1}^{S} L_i$

If α_0 and α_1 are greater than B (N*N+2), then it is better not to encode the block. Instead, the real data is stored after a 2-bit code (00). The procedure *Store_Encoded_Bits* will decide which case is the best (encoding 0's, encoding 1's, or storing the real data) based on E, the minimum of α_0, α_1, and B.

```
Decoder ()
   Read (# of Vectors (R), Vector_Length (C), N);
   Compute_Parameters ();
   For i = 1 to # of segments {
     For j = 1 to # of blocks in i {
       b1b0 = Read_Bits (2);
       Case b1b0
           00 : Read_Bits (N* N);
           01 : b_type = Read_Bits (1);
                Fill_Block (j, b_type);
           10 : Decode_Shapes (0);
           11 : Decode_Shapes (1);
       End Case;
     }
    Output_Segment ();
  }
End Decoder;

Decode_Shapes (b)
   Num_Shapes = Read_Bits (2*log2 N -3);
   For j = 1 to Num_Shapes
      Shape_type = Read_Bits (2);
      Case Shape_type
          00 : c = Get_Coordinate ();
               Fill_Point (b,c);
          01 : t = Get_Type ();
               c = Get_Coordinate ();
               d = Get_Distance ();
               Fill_Line(b, t, c,d);
          10 :  t = Get_Type ();
                c = Get_Coordinate ();
                d = Get_Distance ();
                Fill_Triangle(b, t, c,d);
          11 : c1 = Get_Coordinate ();
                c2 = Get_Coordinate ();
                Fill_Rectangle (b,c1,c2);
End Decode_Shapes;
```

Figure 3. Test vectors decoding algorithm.

3. Decoding Process

The decoding process is simple and straightforward. In this work, we assume that an embedded processor on a chip will implement the decoding algorithm. A framework illustrating the details of how the test vectors can be transferred from the embedded processor to the tested parts of the chip has been outlined in [7]. A similar framework can be used for our decoding algorithm.

Figure 3 shows the algorithm of the decoder. It first reads the arguments given by the encoder and computes the parameters needed for the decoding process. These parameters include the number of segments, the number

Table 6. Compression results of the proposed scheme for various block sizes.

			Block 8x8 Cmp. Ratio			Block 16x16 Cmp. Ratio			
Circuit	Scan Size	No. Vec	1-distance	0-distance	0/1-distance	1-distance	0-distance	0/1-distance	CPU (sec)
c7552	207	73	**37.873**	37.35	37.754	28.661	30.66	33.618	3
c2670	233	44	49.815	50.39	**51.853**	45.416	46.635	47.444	3
s5378	214	97	50.496	49.961	**51.551**	41.418	42.61	44.19	4
s9234	247	105	42.834	42.803	**43.451**	38.249	38.442	38.905	3
s15850	611	94	59.778	**60.898**	60.32	58.81	59.301	59.632	15
s13207	700	233	83.703	83.518	84.145	84.497	84.566	**85.012**	51
s38417	1664	68	46.114	**46.552**	46.497	42.788	43.024	42.47	29

Table 7. Comparison with the techniques by Jas and Touba [7] and Chandra and Chakrabarty [4].

	Proposed Scheme			**Jas and Touba [8]**			**Chandra and Chakrabarty [4].**		
Circuit	Org. Bits	Cmp. Ratio	Cmp. Bits	Org. Bits	Cmp. Ratio	Cmp. Bits	Org. Bits	Cmp. Ratio	Cmp. Bits
c7552	15111	37.873	9388	62721	**42.39**	36134	-	-	-
c2670	10252	51.853	4936	35183	**58.45**	14619	-	-	-
s5378	20758	**51.551**	10057	29850	39.0	18209	23754	40.70	14086
s9234	25935	**43.451**	14666	48906	26.6	35897	39273	43.34	22252
s15850	57434	**60.898**	22458	86151	46.65	45962	76986	47.11	40717
s13207	163100	**85.012**	24446	186200	73.32	49678	165200	74.78	41664
s38417	113152	46.552	60478	247936	**59.06**	101505	164736	44.12	92055

of blocks in a segment and the dimensions of the partial blocks. In order to reconstruct the vectors, each segment has to be stored before sending its vectors to the circuit under test. For each segment, its blocks are decoded one at a time. The first two bits indicate the status of the block as follows:

- 00: the block is not encoded and the following N*N bits are the real data.
- 01: fill the whole block with 0's or 1's depending on the following bit.
- 10: There are shapes that are filled with 0's.
- 11: There are shapes that are filled with 1's.

For those blocks that have shapes, the procedure *Decode_Shapes* is responsible for decoding these shapes. It reads the number of shapes in the block and then for each shape it reads its type and based on this it reads its parameters and fills it accordingly.

After all the blocks in a segment have been decoded, the segment is output to the circuit under test vector by vector.

4. Experimental Results

In order to demonstrate the effectiveness of our scheme, we have performed experiments on a number of the largest ISCAS85 and full-scanned versions of ISCAS89 benchmark circuits. The experiments were run on a Pentium II processor with a speed of 350 MHz and a 32 Mbyte RAM. We have used the test sets generated by MinTest [14], which are highly compacted test sets, that achieve 100% fault coverage of the detectable faults in each circuit. Test cubes were generated from each test set as this has the advantage of keeping unnecessary assignments as x's, which enables higher compression. Then, the test vectors were sorted to maximize the compression. In this work, test vectors were sorted based on a greedy algorithm. Test vectors sorting based on the 0-distance, the 1-distance, and the 0/1-distance was performed. For both the 0-distance and 0/1-distance sorting, the test vector with more 0's was selected as the first vector. However, for the 1-distance sorting, the vector with more 1's was selected as the first vector.

The test sets were partitioned into blocks of sizes 8x8 and 16x16, respectively. Then, the proposed encoding algorithm was applied for each case separately as shown in Table 6. The second column in the table shows the scan size, which is basically the width of a test vector. The third column indicates the number of test vectors in the test set. The *compression ratio* is computed as:

$$Comp.Ratio = \frac{\#Original\ Bits - \#Compressed\ Bits}{\#Original\ Bits} X\,100$$

As can be seen, the effectiveness of the proposed encoding algorithm is clearly demonstrated as high compression ratio was obtained for all the circuits. For most of the circuits, sorting based on the 0/1-distance on an 8x8 block size produced the best results.

The last column in Table 6 shows the total CPU time used for compressing the test vectors based on the two block sizes and based on the three types of distance sorting, i.e. the total CPU time used to produce the best result, which is highlighted in the table.

Based on the compression results in Table 6, our technique achieves an average compression ratio of around **54%** based on highly compacted tests. In Table 7, we compare the compression ratio obtained by our technique with that obtained by the techniques proposed in [7] and [4]. It is important to point out that although the test sets used in our work are different from those used in [7] and [4], they are considerably smaller. As can be seen from the table, for all the compared circuits, our technique achieves significantly higher compression ratio than the technique in [4]. Furthermore, in four of the circuits, out of seven, our technique achieves higher compression ratio than the technique in [7]. It should be observed here that for the three circuits where the technique in [7] achieves higher compression ratio, their original test sets are significantly larger, i.e. they contain much more redundancy, which leads to higher compression ratio. For example, the original test set used in [7] for the circuit c7552 is more than four times larger than the original test set we used.

All the compressed test sets were decoded and verified by fault simulation. The decoding algorithm is very fast and the decoding time for each test set was in fractions of a second.

5. Conclusions

In this paper, a fast and very efficient compression/ decompression scheme for testing systems-on-a-chip has been presented. The technique is based on encoding the test data by geometric shapes. The test data is partitioned into blocks and then each block is encoded separately. To increase the compression ratio, the scheme exploits test vectors reordering, the block size, the type of bit to be encoded, and whether or not to encode the block. Experimental results on ISCAS85 and full-scanned versions of ISCAS89 benchmark circuits demonstrate the effectiveness of the technique in achieving high compression ratio. An average of 54% compression ratio is achieved on highly compacted test sets. In this work, we assumed that the decompression of test data is performed in software by an embedded processor. Hardware implementation of the decompression algorithm will be investigated in future work.

Acknowledgment

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

References

[1] T. Yamaguchi, M. Tilgner, M. Ishida, and D.S. Ha, "An Efficient Method for Compressing Test Data," *Proc. of Int. Test Conference*, pp. 191-199, 1997.

[2] K. Chakrabarty, B.T. Murray, J. Liu, and M. Zhu, "Test Width Compression for Built-In Self-Testing," *Proc. of International Test Conference*, pp. 328-337, 1997.

[3] M. Ishida, D.S. Ha, and T. Yamaguchi, "COMPACT: A Hybrid Method for Compression Test Data," *Proc. of VLSI Test Symposium*, pp. 62-69, 1998.

[4] A. Chandra and K. Chakrabarty, "Test Data Compression for System-On-a-Chip using Golomb Codes," *Proc. of IEEE VLSI Test Symposium*, 2000.

[5] J. Rajski, J. Tyszer, and N. Zaccharia, "Test Data Decompression for Multiple Scan Designs with Boundary Scan," *IEEE Trans. Computers*, pp. 1180-1200, Nov. 1998.

[6] S. Chakradhar and A. Raghunathan, "Bottleneck Removal Algorithm for Dynamic Compaction in Sequencial Circuits," *IEEE Trans. Computer-Aided Design*, 1997.

[7] A. Jas and N.A. Touba, "Using an Embedded Processor for Efficient Deterministic Testing of System-on-a-Chip," *Proc. of IEEE Int. Conf. on Computer Design (ICCD)*, 1999.

[8] A. Jas and N.A. Touba, "Test Vector Decompression via Cyclical Scan Chains and its Application to Testing Core-Based Designs," *Proc. of Int. Test Conf.*, pp. 458-464, 1998.

[9] A. Jas, J.G. Dastidar and N.A. Touba, "Scan Vector Compression/ Decompression Using Statistical Coding," *Proc. of Int. Test Conference*, pp. 458-464, 1998.

[10] A. Jas, K. Mohanram, and N.A. Touba, "An Embedded Core DFT Scheme to Obtain Highly Compressed Test Sets," *Proc. of IEEE Asian Test Symposium*, 1999.

[11] R. Chandramouli, and S. Pateras, "Testing Systems on a Chip," *IEEE Spectrum, pp. 42-47, Nov. 1996*.

[12] Y. Zorian, E.J. Marinissen, and S. Dey, "Testing Embedded-Core Based System Chips," *Proc. of Int. Test Conference*, pp. 130-143, 1998.

[13] G. Gibson et-al, *Digital Compression for Multimedia*, Morgan Kaufmann Publishers, Inc., 1998.

[14] I. Hamzaoglu and J. H. Patel, "Test Set Compaction Algorithms for Combinational Circuits'', *Proc. Int. Conf. Computer-Aided Design*, Nov. 1998.

[15] M. Schulz, E. Trischhler, and T. Sarfert, "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System," *IEEE Trans. Computer-Aided Design*, pp. 126-137, Jan. 1988.

[16] J. Chang and C. Lin, "Test Set Compaction for Combinational Circuits," *IEEE Trans. Computer Aided Design*, pp. 1370-1378, Nov. 1995.

[17] I. Pomeranz, L. Reddy, and S. Reddy, "COMPACTEST: A Method to Generate Compact Test Sets for Combinational Circuits," *Proc. of Int. Test Conference*, pp. 194-203, 1991.

[18] S. Kajihara, I. Pomeranz, K. Kinoshita, and S. Reddy, "Cost-Effective Generation of Minimal Test sets for Stuck-at Faults in Combinational Circuits," *IEEE Trans. Computer Aided Design*, pp. 1496-1504, Dec. 1995.