Test Data Compression Using Efficient Bitmask and Dictionary Selection Methods

Kanad Basu, *Student Member, IEEE*, and Prabhat Mishra, *Senior Member, IEEE*

***Abstract*—Higher circuit densities in system-on-chip (SOC) designs have led to drastic increase in test data volume. Larger test data size demands not only higher memory requirements, but also an increase in testing time. Test data compression addresses this problem by reducing the test data volume without affecting the overall system performance. This paper proposes a novel test data compression technique using bitmasks which provides a substantial improvement in the compression efficiency without introducing any additional decompression penalty. The major contributions of this paper are as follows: 1) it develops an efficient bitmask selection technique for test data in order to create maximum matching patterns; 2) it develops an efficient dictionary selection method which takes into account the bitmask based compression; and 3) it proposes a test compression technique using efficient dictionary and bitmask selection to significantly reduce the testing time and memory requirements. We have applied our method on various test data sets and compared our results with other existing test compression techniques. Our algorithm outperforms existing dictionary-based approaches by up to 30%, giving a best possible test compression of 92%.**

***Index Terms*—Bitmasks, decompression, test compression.**

I. Introduction

In SYSTEM-ON-CHIP (SOC) designs, higher circuit densities have led to larger volume of test data, which demands larger memory requirement in addition to an increased testing time. Test data compression plays a crucial role, reducing the testing time and memory requirements. It also overcomes the automatic test equipment (ATE) bandwidth limitation. Alternatives to external testing include built-in self-test (BIST). However, BIST is not appropriate for logic testing because of its random-resistant fault and bus contention during test application [1], which leads to inadequate test coverage. Other alternatives like bit-flipping [2] and bit fixing [3] provide greater fault coverage, with the disadvantage that structural information has to be provided. Reduction of test data using structural methods like Illinois Scan Architecture (ILS) [4] demands modification of the design. Test data compression algorithms can reduce the test data to a larger degree without facing any of the aforementioned disadvantages.

The overview of a traditional test compression framework is shown in Fig. 1. The original test data is compressed and stored in the memory. Thus, the memory size is significantly reduced.

Manuscript received November 23, 2008; revised February 09, 2009; accepted May 02, 2009. First published August 25, 2009; current version published August 25, 2010. This work was supported in part by NSF CAREER Award 0746261.

The authors are with the Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL 32611-6120 USA (e-mail: kbasu;prabhat@cise.ufl.edu).

Digital Object Identifier 10.1109/TVLSI.2009.2024116

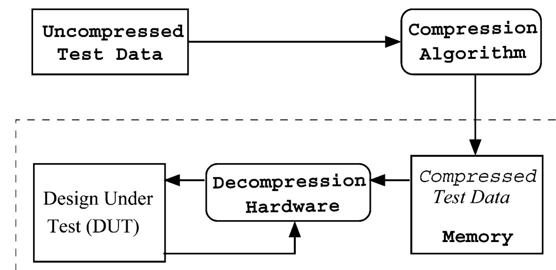

Fig. 1. Test data compression methodology.

An on-chip decoder decodes the compressed test data from the memory and delivers the original uncompressed set of test vectors to the design-under-test (DUT).

Dictionary-based test data compression is a promising approach which has been used by Li *et al.* [1]. Dictionary-based compression techniques are also popular in embedded systems domain since they provide a dual advantage of good compression efficiency as well as fast decompression mechanism. Many recently proposed techniques [5] have tried to improve the dictionary-based compression techniques by considering mismatches. However, the efficiency of these techniques depends on the number of bits allowed to mismatch. It is obvious that if more number of bit changes is allowed, more matching patterns will be generated. However, remembering more bit positions may lead to unprofitable compression. Bitmask based compression [6] addresses this issue by creating more matching patterns with the aid of bitmasks, while taking care of the size of the compressed code. This paper tries to combine the advantages of dictionary based test compression [1] as well as bitmask-based code compression [6].

Bitmask based compression of test data may seem attractive, but it presents various challenges. The primary concern is the presence of don't cares ("X") in the test data set. Since bitmask based compression technique [6] was not designed for data with don't care values, direct application of this technique on test data does not result in a good compression efficiency. We have to determine not only the effective bitmasks, but also a profitable dictionary that produces optimal results. We demonstrate in Section IV that selection of bitmasks and dictionary using existing techniques [1], [6] are not appropriate in case of test data compression using bitmasks. Our approach solves these problems by selecting profitable bitmasks as well as proposing efficient dictionary selection algorithms, to improve the compression efficiency without introducing any additional decompression penalty. Our experimental results demonstrate that our approach produces up to 30% better compression compared to existing dictionary-based approaches [1].

1063-8210/$26.00 © 2009 IEEE

The rest of this paper is organized as follows. Section II describes related work in the area of test data compression. Section III briefly describes bitmask based compression and associated challenges in applying it for test data compression. Section IV presents our compression technique. Section V describes our decompression mechanism. Section VI presents the experimental results. Finally, Section VII concludes this paper.

II. Related Work

Test data compression has manifested itself as a serious problem for a long time. Different compression techniques have been proposed over the years to reduce the test data volume. Some of them are statistical coding [7], run length coding [8], Golomb coding [9], selective Huffman coding [10], run length Huffman coding [11], Tunstall coding [12], LZW coding [13], 9-coded technique [14], heterogeneous compression technique [15], FDR coding [16], multilevel Huffman coding [17], [18] and variable to variable Huffman coding [19]. Reda [20] and Volkerink [21] have shown test data reduction using an on-chip pattern decompression scheme. We have compared our technique with these approaches in Section VI using ISCAS'89 benchmarks obtained from MINTEST ATPG programs.

Dictionary-based compression techniques have been recently used to reduce the test data volume in SOCs. Li *et al.* [1] and Reddy *et al.* [22] used fixed length dictionary entries to reduce test data volume. Dynamic dictionaries along with LZ77 technique has been used by Wolff *et al.* [23]. Wurtenberger *et al.* [24] have proposed a test data compression method by remembering the mismatches with the dictionary entries. A detailed comparison between their approach and ours is provided at the end of this section. We have proposed a bitmask-based compression technique, which renders significantly better results than Li *et al.* [1], as demonstrated in Section VI. Bitmask-based compression was developed by Seong *et al.* [6] for code compression in embedded systems. We have employed a modified version of the bitmask-based compression technique in our test data compression. Our results have demonstrated significant improvement in compression efficiency compared to existing bitmask based compression as shown in Section VI.

Wurtenberger *et al.* [24] have proposed a test data compression technique which is somewhat similar to our approach to remember the mismatches from the dictionary entries. However, there are significant differences between the two approaches. While they [24] try to remember the positions of the bit changes, our method uses bitmasks to account for the mismatch. Due to considering only one bit-fix, [24] looses the opportunity of taking care of multiple mismatches. This is quite evident from our experimental results in Section VI. Our approach performs 10%–30% better compression while uses simpler decompression engine compared to them.

III. Background and Motivation

This section briefly describes bitmask-based code compression [6], and highlight the challenges in employing bitmask-based technique for test data compression.

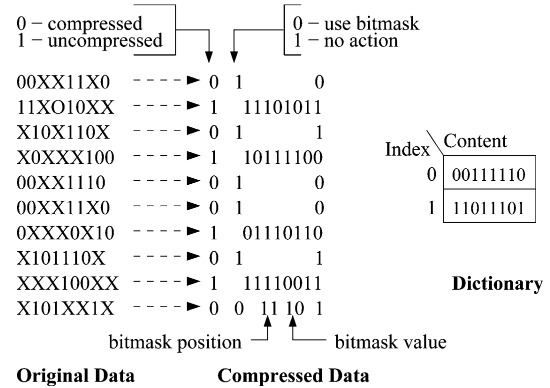

Fig. 2. Bitmask-based test data compression.

A. Bitmask-Based Test Data Compression

Bitmask-based compression is an enhancement on the dictionary-based compression scheme, that helps us to get more matching patterns. In dictionary-based compression, each vector is compressed only if it completely matches with a dictionary entry.

As seen in Fig. 2, we can compress up to six data entries using bitmask based compression. The compressed data is represented as follows. Those vectors that match directly are compressed with 3 bits. The first bit represents whether it is compressed (using 0) or not (using 1). The second bit indicates whether it is compressed using bitmask (using 0) or not (using 1). The last bit indicates the dictionary index. Data that are compressed using bitmask requires 7 bits. The first two bits, as before, represent if the data is compressed, and whether the data is compressed using bitmasks. The next two bits indicate the bitmask position and followed by two bits that indicate the bitmask pattern. For example, the last data vector in Fig. 2 is compressed using a bitmask. The bitmask position is 11, which indicates the fourth even bit position from left. For this case, we have assumed fixed bitmasks, which are always employed on even-bit positions and hence only 2 bits are sufficient to represent the four positions in a 8-bit data. The last bit gives the dictionary index. The bitmask XORed with the dictionary entry produces the original data. More details on the types and positions of bitmasks will be described in Section IV-B in the context of test data compression. In this example, the compression efficiency is 27.5%, based on the following formula expressed as percentage:

$$\text{Compression Efficiency} = \frac{\text{Original Size} - \text{Compressed Size}}{\text{Original Size}}.$$

Since existing approach does not handle don't cares ("X"), in this example we have replaced all don't cares by 1. Note that we could have replaced all don't cares with 0's as well. In that case, it will result in worse compression efficiency of 2.5%. A better compression efficiency can be achieved by selectively replacing the don't cares with "0" or "1" instead of replacing all by 0's (or 1's). It is a major challenge to identify the selective replacement to generate the best possible compression efficiency.

B. Challenges in Bitmask Based Compression

This section outlines various challenges in using bitmask-based approach in test data compression. There are three major

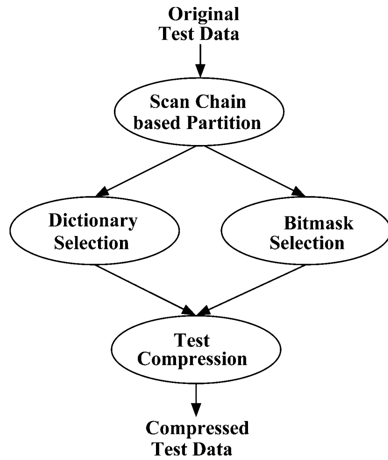

Fig. 3. Bitmask-based test data compression.

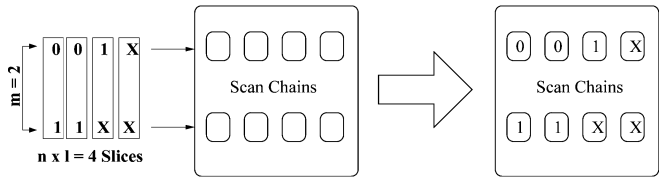

Fig. 4. Dividing test data into scan chains.

challenges in bitmask-based test data compression. The problem is further aggravated since each of these challenges are interdependent and cannot be solved independently.

1) *Dictionary Selection*: A profitable dictionary is to be selected which takes into account the bit savings due to frequency matching as well as bitmasks.
2) *Bitmask Selection*: Appropriate number and type of bitmasks are to be selected for compression.
3) *Don't Care Resolution*: It is necessary to selectively replace each don't care with "0" or "1".

Each of these factors play a crucial role in determining the final compression performance of the algorithm. We begin our discussion with dictionary selection. Frequency of occurrence provided a good base for selection in case of dictionary-based compression, since mismatches would simply be ignored as uncompressed data. However, the scenario is different for bitmask-based compression, where mismatched data could also be compressed using bitmasks. Therefore, care should be taken to select appropriate dictionary entries that account for the bitmasks as well.

Choosing the number and type of bitmasks for compressing a particular data vector is extremely important. There are different types of bitmasks as well. We have to select bitmasks that are most appropriate for compressing a particular group of data. Sections IV-B and IV-C present our bitmask selection and dictionary selection algorithms, respectively.

Another challenge for test data compression is that selective don't care resolution is difficult and it further complicates the dictionary and bitmask selection. Consider the two vectors: "00XX1X10" and "X0X110X0". Although these two vectors have lot of dissimilarities, it is obvious that they will match. This is because a don't care can be matched with a "0" or a "1" according to the matching pattern. Thus a lot more matching can be obtained in case of vectors with don't cares. Therefore, the dictionary selection and bitmask selection algorithms have to be modified in order to incorporate this factor.

IV. Test Data Compression Using Bitmasks

We have developed an efficient test data compression algorithm using bitmasks. Fig. 3 presents our bitmask-based test compression methodology. It has four major steps: 1) divide the input test data based on number of scan chains; 2) dictionary selection; 3) bitmask selection; and 4) test compression using don't care resolution, dictionary and bitmask selection.

Algorithm 1 outlines these steps. The first step divides the uncompressed data set to equal length slices for compression as described in Section IV-A. Next, it tries to determine the profitable bitmasks and dictionary using the procedures described in Sections IV-B and IV-C, respectively. The dictionary selection algorithm generates the dictionary entries while the bitmask selection algorithm provides the number and type of profitable bitmasks. Finally, the test compression is performed using the generated dictionary and bitmasks as illustrated in Section IV-D.

Algorithm 1 Compression Algorithm

Inputs: Test Data
Output: Compressed Test Data and Dictionary
Begin
  1: Divide test data based on scan chains
  2: Select bitmasks
  3: Perform dictionary selection
  4: Compress using selected dictionary and bitmasks
  Return compressed test data and dictionary
End

A. Division of Test Data Into Scan Chains

Once we get the input test data, our next task would be to divide them into scan chains of predetermined length. Let us assume that the test data T_D consists of n test patterns. We divide the scan elements into m scan chains in the best balanced manner possible. This results in each vector being divided into m sub-vectors, each of length, say l. Dissimilarity in the lengths of the sub-vectors are resolved by padding don't cares at the end of the shorter sub-vectors. Thus, all the sub-vectors are of equal length. The m-bit data which is present at the same position of each sub-vector constitute an m-bit slice. If there are n vectors at the beginning, we obtain a total of $n \times l$ m-bit slices, which is our uncompressed data set that needs to be compressed.

Consider a simple example consisting of two $(n = 2)$ test patterns 0011 and 1XXX for a design with two scan chains $(m = 2)$. Therefore, length of each sub-vector $l = 2$. In this case, padding of don't cares is not required. Fig. 4 shows how four slices (XX, 1X, 01, and 01) can be formed with two vectors (001X and 11XX) obtained by scan chain based partitioning of the two original test patterns. These are the four slices that need to be compressed.

Format for Uncompressed Data

Decision (1–bit)	Uncompressed Data

Format for Compressed Data

Decision (1–bit)	# of mask patterns	Mask type	Location	Mask pattern	.....	Dictionary Index

← *Extra bits for considering mismatches* →

Fig. 5. Generic encoding format.

B. Bitmask Selection

Fig. 5 shows the generic encoding formats of bitmask-based compression technique for various number of bitmasks. A compressed data stores information regarding the bitmask type, bitmask location, and the mask pattern itself. The bitmask can be applied on different places on a vector and the number of bits required for indicating the position varies depending on the bitmask type. For instance, if we consider a 32-bit vector, an 8-bit mask applied on only byte boundaries requires 2-bits, since it can be applied on four locations. If we do not restrict the placement of the bitmask, it will require 5 bits to indicate any starting position on a 32-bit vector.

Bitmasks may be sliding or fixed. Fixed bitmasks are referred with the letter **f** while sliding bitmasks are referred with the letter **s**. For example, $2s$ refers to a sliding bitmask of length 2 while $2f$ refers to a fixed bitmask of length 2. A fixed bitmask is one which can be applied to fixed locations, such as byte boundaries. However, sliding bitmasks can be applied anywhere in the test vector. Since the fixed bitmasks can be applied only to fixed locations, the number of positions where they can be applied is significantly less compared to sliding bitmasks. Hence, the number of bits needed to represent them are less than sliding bitmasks.

Seong *et al.* [6] has shown that the profitable bitmasks to be selected for code compression are 2s, 2f, 4s, and 4f. However, in case of test data compression, the last two bitmasks are not profitable. This is because the probability that four corresponding contiguous bits will differ in a pair of test data is only 0.2% [25], which can easily be neglected. Therefore, it is not profitable to use bitmasks of length 4 or higher. Thus, we perform our compression by using 1-bit, 2s or 2f bitmasks. The number of bitmasks selected, which depends on both the test vector length and the dictionary entries, can be determined using the following lemma.

Lemma: The number of bitmasks is dependent on vector length and dictionary entries.

Proof: Let L be the number of dictionary entries and N be the vector length. If y is the number of bitmasks allowed, then in the worst case (when all the bitmasks are $2s$), the number of bits required is shown below which should be less than N.

$$no_bits = 2 + \log_2 L + \log_2 y + y \times (2 + (\log_2 N))$$

The first two bits are required to check whether the data is compressed or not, and if compressed, whether bitmask is used or not. The maximum number of bitmasks allowed can be found by equating the above expression with N, so that the worst case condition holds. Therefore, y can be expressed as follows.

$$y = \frac{N - 2 - \log_2 L}{2 + \log_2 N} - \frac{\log_2 y}{2 + \log_2 N}$$

We can see that it is not easy to compute y since both sides of the equation contain y related terms. Since $y < N$, it is a safe measure to replace the right-most term (the term after the subtraction sign) with 1. Therefore, the following equation for y (floored to the nearest integer) will provide the maximum number of bitmasks that are profitable.

$$y = \frac{N - 2 - \log_2 L}{2 + \log_2 N} - 1$$

C. Dictionary Selection

The dictionary selection algorithm is a critical part in bitmask-based test data compression. Similar to Li *et al.* [1], we used the classical clique partitioning algorithm of graph theory [26]. The clique-partitioning problem basically refers to the breaking up of a graph into several cliques, such that the nodes within one clique are all interconnected. This problem is NP-hard [27], so heuristic approaches are used to solve it in real time.

A graph G is drawn with $n \times l$ nodes, where each node signifies a m-bit test vector. An edge is drawn between two nodes when they are *compatible*. Two nodes are said to be compatible if they meet any one of the following two requirements: 1) for all positions, the corresponding characters in the two vectors are either equal or one of them is a don't care; or 2) two vectors can be matched by predetermined profitable bitmasks. Each edge also contains weight information. The weight is determined based on the number of bits that can be saved by using that edge (direct or bitmask-based matching).

Based on this graph model, we developed three dictionary selection techniques: 1) two-step method (TSM); 2) method using compatible edges (MCE) without edge weights; 3) MCE with edge weights (MEW). Each of these techniques uses a variant of well-known clique partitioning algorithm. The remainder of this section describes these three techniques in detail.

1) Two-Step Method: In TSM, we consider only edges that are formed by direct matching. In other words, the graph will not have any edges corresponding to bitmask-based matching. Then a clique partitioning algorithm [1] is performed on the graph. This is a heuristics-based procedure that selects the node with the largest connectivity and is entered as the first entry to a clique. Now, the nodes connected with it are analyzed, and the node having the largest connectivity among these (and not in the entire graph) is selected. This process is repeated until no node remains to be selected. The entries of the clique are deleted from the graph. The algorithm is repeated until the graph becomes empty. The clique partitioning algorithm is used in MCE and MEW as well.

Since we have a predefined number of dictionary entries, two possibilities may arise. The number of cliques selected may be greater than the predefined number of entries or vice versa. In the latter case, we just need to fill in the dictionary entries with those obtained from clique partitioning. However, if the number

of cliques is larger, we have to select the best dictionary entries as illustrated in Algorithm 2 by considering maximum overall savings using both frequency and bitmasks.

Algorithm 2 Selection of Profitable Dictionary Entries

Inputs: 1. Dictionary entries, S, from clique partitioning
    2. Original Test Data Vectors, V
    3. No. of entries allowed, N
Output: Set of N profitable dictionary entries
Begin
  $ProfitableEntries = \{\}$;
  1: **for** each dictionary entry
    Compute savings by frequency and bitmasks.
  2: **for** count from 1 to N
    2.1 Select the entry D with maximum savings.
    2.2 $ProfitableEntries += D$;
    2.3 $S = S - D$.
    2.4$V = V -$ entries compressed by D.
    2.5 Recompute the savings of S using V
  Return *ProfitableEntries*
End

---

2) Method Using Compatible Edges (MCE) Without Edge Weights: In MCE, weight of all the edges (direct or bitmask-based match) are considered equal. A clique selection algorithm is then performed in the same way as discussed in Section IV-C1.

3) MCE With Edge Weights (MEW): MEW is same as MCE except that we consider edge weights. As indicated earlier, the edge weight is determined based on the number of bits saved if that edge is used for direct or bitmask-based matching. During clique partitioning, instead of connectivity, the total savings by each node is taken into account. The total savings by each node is obtained as the sum of weights of edges originating from that node.

We illustrate how these three methods work using the example test data set in Table I. The resultant graph is shown in Fig. 6. The straight lines in the graph indicate a direct match while the dotted lines signify a match by applying one bitmask. Obviously, the dotted lines will be absent in case of TSM. The dotted lines will have the same weight as the straight lines for MCE. However, they will have different weights in case of MEW. In case of MEW, the weight is determined based on the number of bits saved by using that edge (direct or bitmask match).

When we try to compress these data using TSM, the cliques selected are $\{1, 2, 3\}$ and $\{8, 9, 10\}$. The compression efficiency is obtained as 46.25%. However, when we compress using either MCE or MEW, the cliques selected are $\{4, 5, 6, 7\}$ and $\{1, 2, 3\}$, resulting in improved compression efficiency of 48.75%.

D. Test Compression: An Example

We return to the same example as in Section III-A. Now we try to compress the data using our proposed algorithm in Fig. 7. It can be seen that the don't cares selectively attain values 0 and 1 according to the dictionary entries. The example shows that we are able to compress 9 out of 10 test vectors. The final test data size is 52, which is equivalent to a compression efficiency of 35%, 7.5% more than what we obtained in Section III-A.

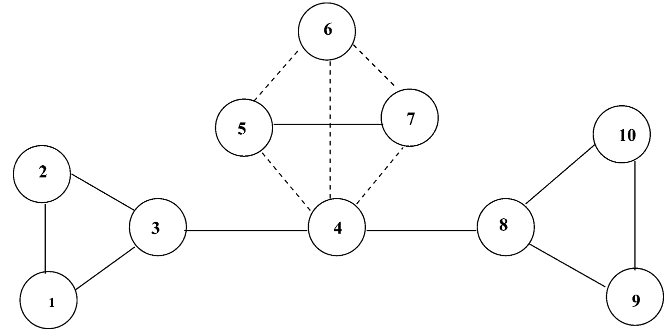

Fig. 6. Graph model for the test data in Table I.

TABLE I
TEST DATA SETS

Data Set	Entry
1	11X001XX01100110
2	1X00X10101100110
3	X1000X0101100110
4	0XXX00X101100110
5	001X011101100110
6	001X101101100110
7	001X011101100110
8	000X0XX101100110
9	XX01X11X01100110
10	XXX101X101100110

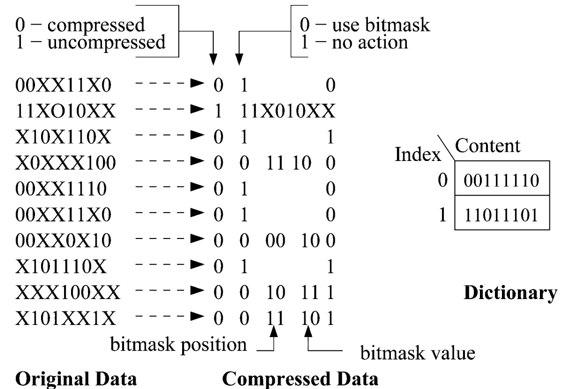

Fig. 7. Compression using bitmasks.

V. DECOMPRESSION MECHANISM

We have proposed the design of a decompression engine (DCE), shown in Fig. 8, that can easily handle bitmasks and provide fast decompression. The design of our engine is based on the one cycle decompression engine proposed by Seong *et al.* [6]. The most important feature is the introduction of XOR gate in addition to the decompression scheme for dictionary based compression. The decompression engine generates a test data length bitmask, which is then XORed with the dictionary entry. The test data length bitmask is created by applying the bitmask on the specified position in the encoding. The generation of bitmask is done in parallel with dictionary access, thus

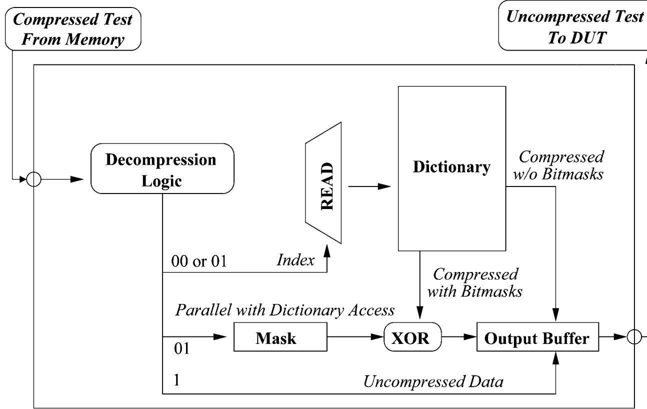

Fig. 8. Decompression engine for bitmask-based encoding.

reducing additional penalty. The DCE can decode more than one compressed data in one cycle.

Algorithm 3 provides an overview of our decompression procedure. The decompression engine takes the compressed vector as input. It checks the first bit to see whether the data is compressed. If the first bit is "1" (implies uncompressed), it directly sends the uncompressed data to the output buffer. On the other hand, if the first bit is a "0", it implies this is a compressed data. Now, there are two possibilities in this scenario. The data may be compressed directly using dictionary entry or may have used bitmasks. The decompression engine will operate differently in these two cases.

Algorithm 3 Decompression of compressed test data

Inputs: Compressed test vectors, length of uncompressed test vector l, dictionary entries
Output: Uncompressed (original) test vectors
Begin
1: Let p be the first bit of the compressed string
2: if $p == 1$
The remaining bits are the uncompressed string
Go to step 9
3: Let q be the second bit of the compressed string
4: **if** $q == 1$
The remaining bits correspond to dictionary index
It is a direct match, read the dictionary entry
Go to step 9
5: Read respective bits for mask values and positions
6: Compose final bitmask using the data from step 5
7: Read the dictionary entry based on dictionary index
8: Perform XOR of dictionary entry with final bitmask to generate the uncompressed string.
9: Send the uncompressed string to the DUT
End

The DCE distinguishes between the two by looking at the second bit. If the second bit is "0", it signifies that it has been matched using only dictionary entry. The index of the dictionary entry is used to read from the dictionary, and the content is transferred to the output buffer. On the other hand, if the second bit is "1", the engine knows that the data has been compressed using bitmasks. Now, the DCE extracts the number and positions of the bitmasks from the compressed data. It extracts the index corresponding to the dictionary entry as well. It then proceeds to create a bitmask. Initially, it creates a vector of the length of the uncompressed vector. It then finds the positions and types of bitmasks and inserts the bitmasks in those positions. Along with the bitmask generation, it does a parallel dictionary lookup in the same way as traditional dictionary based compression. The two results are now XORed. The final result is sent to the output buffer.

TABLE II
COMPRESSION EFFICIENCY OF OUR THREE COMPRESSION SCHEMES

	Best Compression Efficiency(%)		
Circuits	TSM	MCE	MEW
s9234	86.7	87.6	87.5
s13207	91.9	92	92
s15850	88.7	88	88.1
s38417	71.5	73.6	74
s38584	76.4	76.6	76.5

VI. Experiments

In this section, we compare the compression performance and decompression overhead of our approach with other popular test compression algorithms. Section VI-A compares the compression efficiency, while Section VI-B presents the decompression overhead. To demonstrate the usefulness of our method, we have applied it on the tests which were obtained from the MINTEST ATPG program [28] for the five largest ISCAS'89 circuits.

A. Compression Performance

Table II shows the compression performance of TSM, MCE, MEW using the five largest ATPG programs. As expected, MCE and MEW generated better compression by exploiting compatible edges and thereby selecting larger cliques. However, larger clique may not always generate better results when TSM is able to directly match with a large number of test vectors. For example, in case of s15850, TSM has performed better than others due to the fact that almost 88% of the test vectors are matched directly in TSM, whereas around 65% of the test vectors are directly matched using MCE and MEW.

To compare the performance of MEW and MCE, we analyze s38417. As expected, for s38417 in Table II, MEW performs better than MCE, which in turn performs better than TSM. The is due to the fact that a large number of test vectors can be compressed using one or two bitmasks. The number of directly matched test vectors for TSM, MCE, and MEW are 54%, 38%, and 37%, respectively. However, when we consider matching by one or two bitmasks, TSM only covers 8%, while MCE and MEW matches 31.5% and 34% respectively. This huge disparity in bitmask-based matches makes MEW the best method followed by MCE. In some cases, like s9234, MCE is seen to perform 0.1% better than MEW because using MCE, almost 91% of the test vectors can be compressed either directly or using one or two bitmasks. However, for MEW, the number reduces to 90.4%. This minor disparity is due to the fact that we use heuristic method for clique partitioning.

Table III compares the run time information for compression of the test cases obtained from the five largest ATPG programs.

TABLE III
COMPRESSION TIME OF OUR THREE COMPRESSION SCHEMES

	Time taken in seconds		
Circuits	TSM	MCE	MEW
s9234	1	1	1
s13207	12	17	17
s15850	1	1	2
s38417	3	4	4
s38584	6	13	14

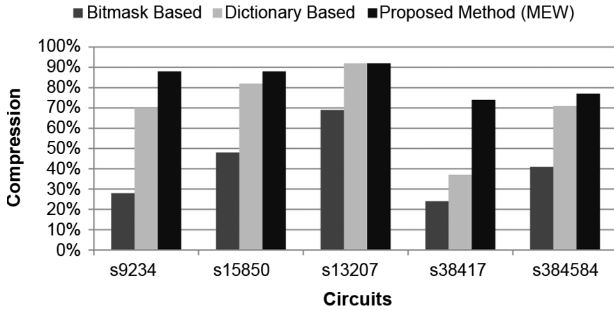

Fig. 9. Compression efficiency for different circuits.

TABLE IV
COMPARISON OF COMPRESSION EFFICIENCY FOR DIFFERENT CIRCUITS

	Compression Efficiency 64 scan chains(%)		Compression Efficiency 128 scan chains(%)	
Circuit	Dictionary based	Our approach	Dictionary based	Our approach
s9234	67.46	75.73	70.12	87.54
s13207	85.43	84.88	91.53	92.01
s15850	75.88	79.28	81.98	88.12
s38417	42.73	65.29	36.75	74.00
s38584	70.77	72.54	70.77	76.51

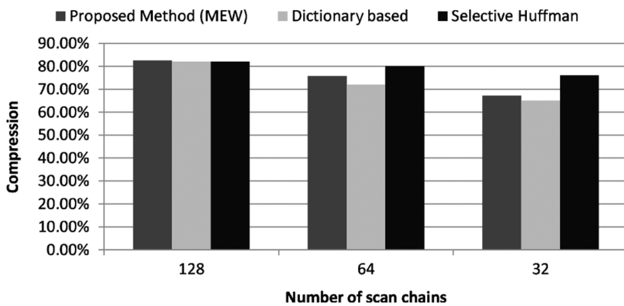

Fig. 10. Compression performance of s15850 for different scan chain lengths and 64 dictionary entries.

It can be seen that although MEW is expected to give the best compression performance, it takes the largest compression time.

In the remainder of this section, our proposed approach refers to MEW. We compare the compression performance of our approach with those obtained by employing the algorithms of Li *et al.* [1] and Seong *et al.* [6]. Fig. 9 shows the comparison between the three techniques. Here, we have considered only 128 scan chains. We have compared the test data obtained from the 5 largest circuits using 128 dictionary entries.

The first bar represents the compression if bitmask-based compression technique [6] is directly applied for compressing the test data. The second bar represents the compression using dictionaries of fixed-length indices [1]. The third bar, which represents our method, gives the best compression efficiency.

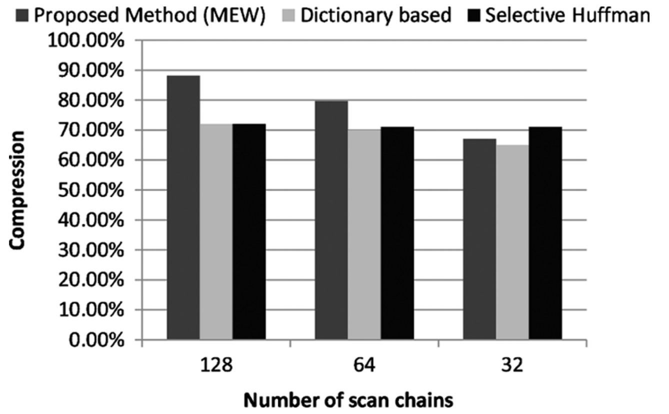

Fig. 11. Compression performance of s15850 for different scan chain lengths and 128 dictionary entries.

TABLE V
COMPRESSION EFFICIENCY FOR DIFFERENT CIRCUITS

		Best Compression Efficiency(%)	
Circuit	Dictionary Entries	Bitmask based Compression	Our approach
c1355	16	34.27	60.28
c499	16	59.15	62.06
c6288	16	33.64	66.35

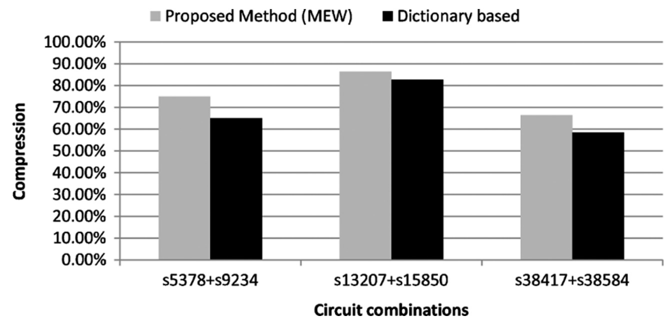

Fig. 12. Compression performances of different circuit combinations with our approach and dictionary-based compression.

TABLE VI
COMPARISON OF COMPRESSION EFFICIENCY FOR DIFFERENT CIRCUITS

	Compression Efficiency 64-bit scan chains(%)		Compression Efficiency 128-bit scan chains(%)	
Circuit	[24]	Our approach	[24]	Our approach
s9234	52	75.73	45	87.54
s13207	75	84.88	81	92.01
s15850	63	79.28	64	88.12
s38417	48	65.29	37	74.00
s38584	62	72.54	67	76.51

The key point to be noted here is the difference between the applications of our approach and the algorithm of Seong *et al.* [6]. Although both rely on bitmasks, due to differences in the dictionary selection algorithm, our approach outperforms the conventional bitmask based approach [6] by 30% to 60%. It is also evident that our approach significantly outperforms (up to 30%) the existing dictionary based compression algorithm [1]. This can be attributed to the introduction of bitmasks in our approach. Bitmasks allow bit changes (which was absent in the dictionary-based compression [1]) and thus matches more vectors.

TABLE VII
PERFORMANCE OF DIFFERENT COMPRESSION SCHEMES USING MINTEST TEST DATA

	Best Compression Efficiency(%)												
	Existing Methods										Our Approach		
Circuits	Golomb [9]	Selective Huffman [10]	RL Huff-man [11]	Tunstall [12]	LZW [13]	9-coded [14]	Var2Var Huffman [19]	Hetero. comp. [15]	Multilevel Huffman [17,18]	FDR [16]	TSM	MCE	MEW
s9234	45	54	42	59	71	55	67	-	61	61	87	88	88
s13207	80	30	16	34	82	85	91	94	89	88	92	92	92
s15850	63	38	32	45	76	71	80	44	75	72	89	88	88
s38417	28	45	44	68	71	63	64	47	64	65	71	74	74
s38584	57	40	43	50	75	69	72	81	73	64	76	77	77

TABLE VIII
TEST COMPRESSION TIME FOR DIFFERENT TECHNIQUES

	Time needed in seconds	
Circuits	Selective Huffman	Our approach
s9234	2.6	1
s13207	2.2	17
s15850	3.5	2
s38417	15.8	4
s38584	17.8	14

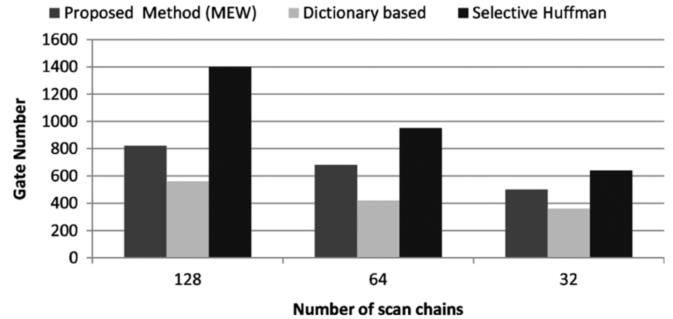

Fig. 13. Gate numbers for decompression using the three methods for 64 dictionary entries.

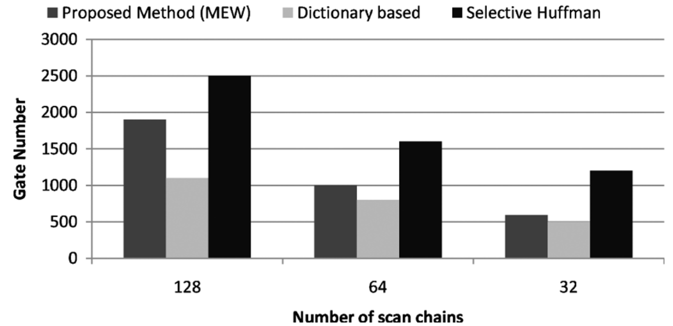

Fig. 14. Gate numbers for decompression using the three methods for 128-bit dictionary entries.

We now compare the compression efficiencies obtained by Li *et al.* [1] with our approach for the test cases obtained from the 5 ATPG programs using both 64- and 128-bit scan chains. Table IV demonstrates that in almost all cases, our approach provides better compression efficiency than dictionary-based compression [1].

We now compare the performance of our approach against the selective Huffman coding [10] on test vectors obtained from one of the largest circuits, that is s15850. Fig. 10 presents the results of comparison of the two compression schemes for 64 dictionary entries and variable number of scan chains. Fig. 11 compares the two approaches for 128 dictionary entries and variable number of scan chains.

It can be seen that our approach works better with greater number of scan chains. The reason is quite obvious. With smaller number of scan chains and larger dictionary entries, less number of bit vectors can be profitably compressed using bitmasks. It is important to note that our approach performs better than the dictionary based compression scheme in all cases. Although the selective Huffman coding gives better compression performance, it has significant decompression area and performance overhead as discussed in Section VI-B.

So far we have demonstrated that our approach outperforms the existing approaches for test data compression in the presence of don't cares in the test data. We have also applied our algorithm in test data sequences which do not have don't cares. These test data are obtained from some of the MINTEST ATPG programs [28]. We have compared results of our approach with those obtained when compressed using the existing bitmask-based compression method [6]. Table V presents the results of test data compression (without don't cares). As expected, our approach provides better compression than the bitmask-based compression [6].

In our next experiment, we combine pairs of test data sets that have nearly the same number of scan chains and then compress them. This is done in order to test the performance of our compression algorithm on really huge data sets which are formed by the combination of a pair of test data. We then compare our approach with the dictionary-based compression algorithm [1]. We create three sets of test data by combining the test data of the circuits in each set. The sets are $\{s5378, s9234\}$, $\{s13207, s15850\}$, and $\{s38417, s38584\}$. The results are shown in Fig. 12. It can be seen that our approach performs up to 25% better than Li *et al.* [1].

We have compared our approach with the algorithm proposed by Wurtenberger *et al.* [24] that compressed test data by remembering the mismatches from the dictionary entries. The comparison results are given in Table VI for the ISCAS'89 benchmarks. Our approach performs 10%–30% better compression than them.

Table VII presents the comparison of our approaches (TSM, MCE, and MEW) with various existing compression techniques. Table VIII compares our test compression time with that of selective Huffman coding.

B. Decompression Overhead

This section compares area overhead of our DCE with other approaches. We have used the DCE to decompress the compressed data obtained from s15850. The results are compared with those obtained using dictionary based compression [1] and selective Huffman-based compression [10].

Fig. 13 shows the results for 64 dictionary entries while Fig. 14 presents it for 128 dictionary entries. In both cases our method requires less area (number of gates) compared to selective Huffman-based encoding. However, it requires larger area compared to dictionary-based compression. This is quite obvious since dictionary based compression has only few states, whereas we need more states due to use of bitmasks. Our decompression engine has three primary components that contribute to the area overhead: SRAM to store the dictionary, bitmask generation logic, and final XOR. The last two components are not needed in dictionary-based compression.

VII. Conclusion

This paper presented a test compression algorithm that combines the advantages of dictionary-based compression [1] and bitmask-based compression [6]. This paper developed efficient bitmask and dictionary selection techniques for test data compression in order to create maximum matching patterns in the presence of don't cares. Our test compression technique used the dictionary and bitmask selection methods to significantly reduce the testing time and memory requirements. We have applied our algorithm on various benchmarks and compared our results with existing test compression techniques. Our algorithm outperforms existing dictionary based compression [1] by up to 30%, giving a best possible compression of 92%. Our approach also generates up to 60% improvement in compression efficiency compared to bitmask-based compression [6] without introducing any additional performance or area overhead.

Acknowledgment

The authors would like to thank Prof. K. Chakrabarty for providing the test data for ISCAS'89 benchmarks obtained from MINTEST ATPG programs.

References

[1] L. Li, K. Chakrabarty, and N. Touba, "Test data compression using dictionaries with selective entries and fixed-length indices," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 8, no. 4, pp. 470–490, 2003.

[2] H. Wunderlich and G. Kiefer, "Bit-flipping BIST," in *Proc. Int. Conf. Comput.-Aided Des.*, 1996, pp. 337–343.

[3] N. Touba and E. McCluskey, "Altering a pseudo-random bit sequence for scan based bist," in *Proc. Int. Test Conf.*, 1996, pp. 167–175.

[4] F. Hsu, K. Butler, and J. Patel, "A case study on the implementation of Illinois scan architecture," in *Proc. Int. Test Conf.*, 2001, pp. 538–547.

[5] M. Ros and P. Sutton, "A hamming distance based VLIW/EPIC code compression technique," in *Proc. Compilers, Arch., Synth. Embed. Syst.*, 2004, pp. 132–139.

[6] S. Seong and P. Mishra, "Bitmask-based code compression for embedded systems," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 27, no. 4, pp. 673–685, Apr. 2008.

[7] M.-E. N. A. Jas, J. Ghosh-Dastidar, and N. Touba, "An efficient test vector compression scheme using selective Huffman coding," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 22, no. 6, pp. 797–806, Jun. 2003.

[8] A. Jas and N. Touba, "Test vector decompression using cyclical scan chains and its application to testing core based design," in *Proc. Int. Test Conf.*, 1998, pp. 458–464.

[9] A. Chandra and K. Chakrabarty, "System on a chip test data compression and decompression architectures based on Golomb codes," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 20, no. 3, pp. 355–368, Mar. 2001.

[10] X. Kavousianos, E. Kalligeros, and D. Nikolos, "Optimal selective Huffman coding for test-data compression," *IEEE Trans. Computers*, vol. 56, no. 8, pp. 1146–1152, Aug. 2007.

[11] M. Nourani and M. Tehranipour, "RL-Huffman encoding for test compression and power reduction in scan applications," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 10, no. 1, pp. 91–115, 2005.

[12] H. Hashempour, L. Schiano, and F. Lombardi, "Error-resilient test data compression using Tunstall codes," in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Syst.*, 2004, pp. 316–323.

[13] M. Knieser, F. Wolff, C. Papachristou, D. Weyer, and D. McIntyre, "A technique for high ratio LZW compression," in *Proc. Des., Autom., Test Eur.*, 2003, p. 10116.

[14] M. Tehranipour, M. Nourani, and K. Chakrabarty, "Nine-coded compression technique for testing embedded cores in SOCs," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 13, pp. 719–731, Jun. 2005.

[15] L. Lingappan, S. Ravi, A. Raghunathan, N. K. Jha, and S. T. Chakradhar, "Test-volume reduction in systems-on-a-chip using heterogeneous and multilevel compression techniques," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 25, no. 10, pp. 2193–2206, Oct. 2006.

[16] A. Chandra and K. Chakrabarty, "Test data compression and test resource partitioning for system-on-a-chip using frequency-directed run-length (FDR) codes," *IEEE Trans. Computers*, vol. 52, no. 8, pp. 1076–1088, Aug. 2003.

[17] X. Kavousianos, E. Kalligeros, and D. Nikolos, "Multilevel-Huffman test-data compression for IP cores with multiple scan chains," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 16, no. 7, pp. 926–931, Jul. 2008.

[18] X. Kavousianos, E. Kalligeros, and D. Nikolos, "Multilevel Huffman coding: An efficient test-data compression method for IP cores," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 26, no. 6, pp. 1070–1083, Jun. 2007.

[19] X. Kavousianos, E. Kalligeros, and D. Nikolos, "Test data compression based on variable-to-variable Huffman encoding with codeword reusability," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 27, no. 7, pp. 1333–1338, Jul. 2008.

[20] S. Reda and A. Orailoglu, "Reducing test application time through test data mutation encoding," in *Proc. Des. Autom. Test Eur.*, 2002, pp. 387–393.

[21] E. Volkerink, A. Khoche, and S. Mitra, "Packet-based input test data compression techniques," in *Proc. Int. Test Conf.*, 2002, pp. 154–163.

[22] S. Reddy, K. Miyase, S. Kajihara, and I. Pomeranz, "On test data volume reduction for multiple scan chain design," in *Proc. VLSI Test Symp.*, 2002, pp. 103–108.

[23] F. Wolff and C. Papachristou, "Multiscan-based test compression and hardware decompression using LZ77," in *Proc. Int. Test Conf.*, 2002, pp. 331–339.

[24] A. Wurtenberger, C. Tautermann, and S. Hellebrand, "Data compression for multiple scan chains using dictionaries with corrections," in *Proc. Int. Test Conf.*, 2004, pp. 926–935.

[25] K. Basu and P. Mishra, "A novel test-data compression technique using application-aware bitmask and dictionary selection methods," in *Proc. ACM Great Lakes Symp. VLSI*, 2008, pp. 83–88.

[26] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*. Boston, MA: MIT Press, 2001.

[27] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: Freeman, 1979.

[28] I. Hamzaoglu and J. Patel, "Test set compaction algorithm for combinational circuits," in *Proc. Int. Conf. Comput.-Aided Des.*, 1998, pp. 283–289.

Kanad Basu (S'08) received the B.E. degree in electronics and telecommunication engineering from Jadavpur University, Calcutta, India, in 2007. He is currently pursuing the Ph.D. degree from the Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL.

His research interests include the area of post silicon validation, test compression, and system verification.

Prabhat Mishra (S'00–M'04–SM'08) received the B.E. degree from Jadavpur University, Calcutta, India, in 1994, the M.Tech. degree from the Indian Institute of Technology, Kharagpur, India, in 1996, and the Ph.D. degree from the University of California, Irvine, in 2004, all in computer science.

He spent several years with various semiconductor and design automation companies including Texas Instruments, Synopsys, Intel, and Freescale (Motorola). He is currently an Assistant Professor with the Department of Computer and Information Science and Engineering, University of Florida, Gainesville. His research interests include design automation of embedded systems, reconfigurable architectures, and functional verification. He is the coauthor of the book *Functional Verification of Programmable Embedded Architectures* (Kluwer, 2005). He is also the coeditor of the book *Processor Description Languages* (Morgan Kaufmann, 2008).

Dr. Mishra currently serves as Program Chair of IEEE High Level Design Validation and Test (HLDVT) workshop, Information Director of ACM *Transactions on Design Automation of Electronic Systems* (TODAES), Guest Editor of Springer International Journal of Parallel Programming (IJPP), as a program/organizing committee member of several ACM and IEEE conferences, and as a reviewer of many premier journals, conferences and workshops. He is a professional member of ACM. His research has been recognized by various awards, including an NSF CAREER Award in 2008, the EDAA Outstanding Dissertation Award in 2005 and the CODES+ISSS Best Paper Award in 2003.