Relationship Between Entropy and Test Data Compression

Kedarnath J. Balakrishnan, *Member, IEEE*, and Nur A. Touba, *Senior Member, IEEE*

***Abstract*—The entropy of a set of data is a measure of the amount of information contained in it. Entropy calculations for fully specified data have been used to get a theoretical bound on how much that data can be compressed. This paper extends the concept of entropy for incompletely specified test data (i.e., that has unspecified or don't care bits) and explores the use of entropy to show how bounds on the maximum amount of compression for a particular symbol partitioning can be calculated. The impact of different ways of partitioning the test data into symbols on entropy is studied. For a class of partitions that use fixed-length symbols, a greedy algorithm for specifying the don't cares to reduce entropy is described. It is shown to be equivalent to the minimum entropy set cover problem and thus is within an additive constant error with respect to the minimum entropy possible among all ways of specifying the don't cares. A polynomial time algorithm that can be used to approximate the calculation of entropy is described. Different test data compression techniques proposed in the literature are analyzed with respect to the entropy bounds. The limitations and advantages of certain types of test data encoding strategies are studied using entropy theory.**

***Index Terms*—Entropy theory, linear feedback shift register (LFSR) reseeding, test data compression.**

I. Introduction

ONE of the main challenges in very large scale integration testing has been the rapid increase in test data volume especially in system-on-a-chip designs [2]. More test data volume leads to higher tester memory requirements and longer test times when the testing is performed using conventional external testers, i.e., automated test equipment. This directly affects the test costs of a chip. Reducing the test data volume by compression techniques is an attractive approach for dealing with this problem. The test data are stored in compressed form on the tester and transferred to the chip where it is decompressed using an on-chip decompressor.

Several test data compression schemes have been proposed in the literature. Some of these schemes are applicable to both test stimuli and test response, whereas others consider compression of test stimuli only or test response only. For compressing test response, lossy techniques can be used that achieve higher compression while still preserving the fault coverage. However, lossless compression is needed for the test stimuli to preserve the fault coverage. In this paper, we study the compression of test stimuli only, and the terms test vector and test data are used interchangeably with test stimuli.

Manuscript received May 9, 2005; revised September 13, 2005, January 25, 2006, and May 30, 2006. This work was supported in part by Intel Corporation and in part by the National Science Foundation under Grant CCR-0306238. This paper was recommended by Associate Editor S. Hellebrand.

K. J. Balakrishnan was with the System LSI Department, NEC Laboratories America, Princeton, NJ 08540 USA. He is now with the DFT/ATPG Group, Advanced Micro Devices, Inc., Austin, TX 78741 USA (e-mail: kedarnath.balakrishnan@amd.com).

N. A. Touba is with the Department of Electrical and Computer Engineering, University of Texas, Austin, TX 78712 USA (e-mail: touba@ece.utexas.edu).

Digital Object Identifier 10.1109/TCAD.2006.882600

In general, compression schemes can be regarded as an encoding of data, usually into a smaller size than the original data to achieve compression. In this paper, encoded data or compressed data mean the same, i.e., the output of the compression scheme. The sets of bits operated upon are called blocks. Test vector compression schemes can be classified into different categories depending on how test data are handled by the scheme. If the scheme encodes a fixed number of input bits into a fixed number of encoded bits, it belongs to the "fixed-to-fixed" category. Similarly, schemes that encode a fixed number of input bits to a variable number of encoded bits are classified under the fixed-to-variable category. The two other possible categories are "variable-to-fixed" schemes that encode a variable number of input bits into a fixed number of encoded bits and "variable-to-variable" schemes that encode a variable number of input bits into a variable number of encoded bits. These different categories and the compression schemes that fall under each category are discussed further as follows.

A. Fixed-to-Fixed Schemes

An example of a fixed-to-fixed scheme is conventional linear feedback shift register (LFSR) reseeding [3], where each fixed-size test vector is encoded as a smaller fixed-size LFSR seed. Techniques that use combinational expanders with more outputs than inputs to fill more scan chains with fewer tester channels each clock cycle fall into this category. These techniques include using linear combinational expanders such as broadcast networks [4] and XOR networks [5], as well as nonlinear combinational expanders [6], [7]. If the size of the input blocks is n bits and the size of the encoded blocks is b bits, then there are 2^n possible symbols (input block combinations) and 2^b possible codewords (encoded block combinations). Since b is less than n, obviously not all possible symbols can be encoded using a fixed-to-fixed code. If S_{dict} is the set of symbols that can be encoded (i.e., are in the "dictionary") and S_{data} is the set of symbols that occur in the input data, then if $S_{\mathrm{data}} \subseteq S_{\mathrm{dict}}$, it is a complete encoding; otherwise, it is a partial encoding. For LFSR reseeding, it has been shown in [3] that if b is chosen to be 20 bits larger than the maximum number of specified bits in any n-bit block of the input data, then the probability of not having a complete encoding is less than 10^{-6}. The technique in [6] constructs the nonlinear combinational expander so that it will implement a complete encoding. For techniques that do not have a complete encoding, there are two alternatives.

Either the automatic test pattern generation (ATPG) process should be constrained, so that it only generates test data that are contained in the dictionary as in [5], or the dictionary for symbols that are not contained in it like in [7] should be bypassed. Bypassing the dictionary requires adding an extra bit to each codeword to indicate whether it is coded data or not.

B. Fixed-to-Variable Schemes

Huffman codes are the best example of fixed-to-variable coding schemes. The idea with a Huffman code is to encode symbols that occur more frequently with shorter codewords and symbols that occur less frequently with longer codewords. A method was shown in [8] to construct the code in a way that minimizes the average length of a codeword. The problem with a Huffman code is that the decoder grows exponentially as the size of the input blocks is increased. In [9] and [10], the idea of a selective Huffman code was introduced, where partial coding is used and the dictionary is selectively bypassed. This allows larger block sizes to be efficiently used.

C. Variable-to-Fixed Schemes

Test vector compression schemes based on conventional run-length codes belong to the variable-to-fixed category. A method for encoding variable-length runs of zeroes using fixed-size blocks of encoded data was proposed in [11]. The Lempel–Ziv (LZ)-based coding methods like LZ77 [12] and LZW [13] also fall in this category. Both the LZ methods are dictionary-based methods, although the difference is in how the dictionary is formed. In LZ-based techniques, patterns within a sliding window are replaced with pointers to matching previously seen patterns.

D. Variable-to-Variable Schemes

Several techniques that use run-length codes with a variable number of bits per codeword have been proposed including using Golomb codes [14], frequency directed codes [15], and variable-length input Huffman compression (VIHC) codes [16]. All these belong to the variable-to-variable category. One of the difficulties with variable-to-variable codes is synchronizing the transfer of data from the tester. All of the techniques that have been proposed in this category require the use of a synchronizing signal going from the on-chip decoder back to the tester to tell the tester to stop sending data at certain times while the decoder is busy. Fixed-to-fixed codes do not have this issue because the data transfer rate from the tester to the decoder is constant.

This paper investigates the fundamental limits of test data compression, i.e., it looks to answer the question, "How much can we compress test data?" This is done by looking at the entropy of test data and its relation to the compression limit. The idea of using entropy for calculating the test data compression limits was first studied in [17]. However, in that paper, the entropy calculations were limited to a special case related to frequency-directed run-length codes where all the don't cares were specified as zeroes. Moreover, only one way of partitioning the input data into symbols was considered. Most of the compression schemes proposed in literature take advantage of the presence of unspecified or don't care bits in test data. Different ways of specifying the don't cares and different symbol sets will lead to different entropies for the same test set. In this paper, we investigate entropy including the additional degrees of freedom that were not explored in [17]. In this paper, a procedure for approximately calculating the minimum entropy of a test set for fixed-symbol-length schemes over all possible ways of specifying the don't cares is described. Also, the relationship between entropy and symbol partitioning is studied. Preliminary results were published in [18]; however, note that there was an incorrect claim in [18] of calculating the exact entropy for fixed-length symbols. It is shown here that the entropy is calculated to within an additive constant bound from the exact entropy.

Using entropy theory, it is possible to derive theoretical limits on the compression that can be achieved using various types of coding techniques. This is useful in identifying how much room for improvement there is for the various test data compression techniques that have been proposed. It is also useful to identify the compression techniques that have a lot of room for improvement and offer scope for fruitful research.

The paper is organized as follows: In Section II, the relationship of symbol partitioning and don't care filling to entropy is described. For fixed-symbol-length schemes, a greedy algorithm is described for specifying the don't cares in a way that tries to minimize the entropy. This algorithm is shown to be within an additive constant error with respect to the exact entropy. In Section III, the relationship between symbol length and maximum compression is discussed. In Section IV, different compression schemes that have been proposed in the literature are compared with their entropy limits. In Section V, the compression limits for LFSR reseeding are described. Conclusions are in Section VI.

II. Entropy Analysis for Test Data

Entropy is a measure of the disorder in a system. The entropy of a set of data is related to the amount of information that it contains, which is directly related to the amount of compression that can be achieved. Entropy is equal to the minimum average number of bits needed to represent a codeword and hence presents a fundamental limit on the amount of data compression that can be achieved [19]. The entropy for a set of test data depends on how the test data are partitioned into symbols and how the don't cares are specified. These two degrees of freedom are investigated in this section.

A. Partitioning Test Data Into Symbols

For fixed-to-fixed and fixed-to-variable codes, the test data are partitioned into fixed-length symbols (i.e., each symbol has the same number of bits). The entropy for the test data will depend on the symbol length. Different symbol lengths will have different entropies. For a given symbol length, the entropy for the test data can be calculated and will give a theoretical limit on the amount of compression that can be achieved by any fixed-to-fixed or fixed-to-variable code that uses that symbol length. This is illustrated by the following example.

TABLE I
TEST SET DIVIDED INTO 4-BIT BLOCKS

Vector1	1111	0001	0011	0000	1100	0111
Vector2	0000	0011	0001	0111	0100	0000
Vector3	0001	0111	0111	1100	1100	0000
Vector4	0011	0000	0000	0100	1100	0111

TABLE II
PROBABILITY TABLE FOR SYMBOL LENGTH OF 4

i	Symbol x_i	Frequency	Probability p_i
1	0000	6	0.2500
2	0111	5	0.2083
3	1100	4	0.1667
4	0011	3	0.1250
5	0001	3	0.1250
6	0100	2	0.0833
7	1111	1	0.0416
Entropy = 2.64 Max. Compression = 34.0 %			

TABLE III
PROBABILITY TABLE FOR SYMBOL LENGTH OF 6

i	Symbol x_i	Frequency	Probability p_i
1	000000	4	0.2500
2	010011	2	0.1250
3	000111	2	0.1250
4	111100	1	0.0625
5	001111	1	0.0625
6	110001	1	0.0625
7	011101	1	0.0625
8	000101	1	0.0625
9	110111	1	0.0625
10	110011	1	0.0625
11	001100	1	0.0625
Entropy = 3.25 Max. Compression = 45.8 %			

Consider the test set T in Table I, which consists of four test vectors each partitioned into 4-bit blocks. The probability of occurrence of each unique 4-bit symbol is shown in Table II. Note that the term "probability" here refers to the actual frequency of the symbol with respect to the total number of symbols in the data rather than the classical definition of probability as the chance of occurrence. The column "frequency" shows the number of times each symbol appears in the test set, and the column "probability" is calculated by dividing the frequency with the total number of blocks in the test set. The entropy of this test set is calculated from the probabilities of occurrence of unique symbols using the formula $H = -\sum_{i=1}^{n} p_i \cdot \log p_i$, where p_i is the probability of occurrence of symbol x_i in the test set and n is the total number of unique symbols. This entropy is calculated and shown in the last row of Table II. The entropy gives the minimum average number of bits required for each codeword. Thus, the maximum compression that can be achieved is given by (symbol_length − entropy)/(symbol_length), which in this case is equal to $(4 - 2.64)/4 = 34\%$. Now, if the test set is partitioned into 6-bit blocks instead of 4-bit blocks, then the corresponding probabilities of occurrence of unique symbols is shown in Table III, and the entropy is shown in the last row. As can be seen, the entropy is different for 6-bit blocks than it is for 4-bit blocks. The maximum compression in this case is equal to $(6 - 3.25)/6 = 46\%$.

TABLE IV
PROBABILITY TABLE FOR RUNS OF 0'S

i	Symbol x_i	Frequency	Probability p_i
1	1	22	0.5789
2	01	4	0.1053
3	0001	4	0.1053
4	001	3	0.0789
5	0000000001	2	0.0526
6	00001	1	0.0263
7	0000001	1	0.0263
8	000000001	1	0.0263
Entropy = 2.07 Max. Compression = 18.2 %			

For variable-to-fixed and variable-to-variable codes, the test data are partitioned into variable-length symbols (i.e., the symbols can have different numbers of bits). An example of this is a run-length code where each symbol consists of a different size run length. Given the set of variable-length symbols, the entropy can be calculated the same way as for fixed-length symbols. This is illustrated by partitioning the test set shown in Table I into symbols corresponding to different runs of zeroes. The probability of occurrence of each unique symbol is shown in Table IV, and the entropy is shown in the last row. Calculating the maximum compression for variable-length symbols is different than that for fixed-length symbols. For variable-length symbols, the maximum compression is equal to (avg_symbol_length − entropy)/avg_symbol_length. The average symbol length is computed as $\sum_{i=1}^{n} p_i \cdot |x_i|$, where p_i is the probability of occurrence of symbol x_i, $|x_i|$ is the length of symbol x_i, and n is the total number of unique symbols. For this example, the average symbol length is 2.49; thus, the maximum compression is $(2.53 - 2.07)/2.53 = 18\%$. This is the maximum compression that any code that encodes runs of zeroes can achieve for the test data in Table I.

B. Specifying the Don't Cares

While computing entropy for fully specified data is well understood, the fact that test data generally contains don't cares makes the problem of determining theoretical limits on test data compression more complex. Obviously, the entropy will depend on how the don't cares are filled with ones and zeroes since that affects the frequency distribution of the various symbols. To determine the maximum amount of compression that can be achieved, the don't cares should be filled in a way that minimizes the entropy.

While the number of different ways the don't cares can be filled is obviously exponential, for fixed-length symbols, two algorithms are presented here for filling don't cares that aim to result in minimum entropy frequency distribution. The first algorithm, which is called the greedy fill algorithm, is shown to be within an additive constant error with respect to the minimum entropy. The second algorithm is a polynomial time approximate algorithm to reduce the runtime of calculating entropy for longer symbol lengths. For variable-length symbols, the problem is more difficult and remains an open problem. In [20], this problem is discussed for run-length codes, and an optimization method based on simulated annealing is used to find an approximate solution.

x 0 x 0	1 0 x x	1 x x x	0000 - 3	0100 - 3	1000 - 6	1100 - 5	1 0 0 0	1 0 0 0	1 1 1 1
x x 1 x	x 1 x 1	1 1 1 x	0001 - 1	0101 - 3	1001 - 3	1101 - 4	1 1 1 1	1 1 1 1	1 1 1 1
x x 0 0	1 x x 0	x 1 1 1	0010 - 3	0110 - 3	1010 - 5	1110 - 5	1 0 0 0	1 0 0 0	1 1 1 1
x x x x	x 1 x x	1 0 0 0	0011 - 2	0111 - 2	1011 - 4	1111 - 7	1 1 1 1	1 1 1 1	1 0 0 0
	(a)			(b)				(c)	

Fig. 1. Example of specifying don't cares to minimize entropy. (a) Unspecified test set. (b) Frequency of minterms. (c) After Greedy fill.

1) Greedy Fill Algorithm: The greedy fill algorithm is based on the idea that the don't cares should be specified in a way such that the frequency distribution of the patterns becomes as skewed as possible. This algorithm was first described in [10], where the goal was to maximize the compression for Huffman coding. However, it also applies for specifying the don't cares for minimizing entropy. Algorithm 1 shows the pseudocode for the greedy fill algorithm. If the fixed symbol length is n, then the algorithm identifies the minterm, a fully specified symbol of length n (there are 2^n such minterms) that is contained in as many of n-bit symbols of the unspecified test data as possible, which has the highest frequency of occurrence. This minterm is then selected, and all the unspecified n-bit symbols that contain this minterm are specified to match the minterm. The algorithm then proceeds in the same manner by finding the next most frequently occurring minterm. This continues until all the don't cares have been specified.

```
**Algorithm 1:** Greedy Fill Algorithm
**Input:** Test Set $T$ (with unspecified bits), Symbol Length $n$
**Output:** Test Set $T^\#$
GREEDY FILL$(T, n)$
  $M = \{m : m$ is a minterm of size $n\}$;
  $T^\# = T$;
  while ($T^\#$ has unspecified bits)
    foreach $m \in M$
      frequency$[m] = 0$;
    done
    foreach (symbol $s$ of $T^\#$)
      foreach $m \in M$
        if ($s$ contains $m$)
          frequency$[m]$ ++;
        endif
      done
    done
    $m^* =$ MAX FREQUENCY$(M)$;
    foreach (symbol $s$ of $T^\#$)
      if ($s$ contains $m^*$)
        replace $s$ with $m^*$ in $T^\#$;
      endif
    done
    $M = M - \{m^*\}$;
  endwhile
END
```

To illustrate this algorithm, an example test set consisting of four test vectors with 12 bits each is shown in Fig. 1(a). The test vectors are partitioned into 4-bit symbols resulting in a total of 12 4-bit symbols. The number of symbols that contain each of the 16 $(2^4 = 16)$ possible minterms is shown in Fig. 1(b). The most frequently occurring minterm is 1111 as seven of the unspecified symbols [underlined in Fig. 1(a)] contain 1111. Hence, these symbols are specified to 1111. After the first iteration, only five symbols remain unspecified, and the most frequently occurring minterm now is 1000. All the remaining five symbols contain this minterm, and, hence, the algorithm terminates. The resulting test set after filling up the unspecified bits in the above manner is shown in Fig. 1(c).

TABLE V
ENTROPY COMPRESSION LIMITS USING GREEDY FILL ALGORITHM

Circuit	Test Size	Symbol Length			
		4	6	8	10
s5378	23754	52.0 %	56.3 %	59.2 %	63.8 %
s9234	39723	54.1 %	57.4 %	60.7 %	63.7 %
s13207	165200	83.6 %	85.0 %	85.6 %	87.1 %
s15850	76986	68.9 %	70.5 %	71.9 %	73.5 %
s38417	164736	60.2 %	63.2 %	65.3 %	67.7 %
s38584	199104	65.9 %	67.6 %	68.8 %	70.8 %

The problem of finding the minimum entropy frequency distribution is related to a classical problem in algorithms—the set covering problem [21]. Finding the minimum set of minterms that cover all the symbols in the test set is an example of the set covering problem. However, calculating the minimum entropy frequency distribution is much more complex than finding the minimum set cover. Finding the minimum entropy frequency distribution is similar to the haplotype resolution problem in computational biology that was described in [1]. Using the results in [1], we can derive the following theorem that shows that the entropy achieved with the greedy fill algorithm is within an additive constant error of minimum possible entropy. The reader is referred to [1, Th. 2] for the proof.

Theorem 1: Let $f_{\rm OPT}$ be the frequency distribution with minimum entropy. If f_G is the frequency distribution obtained using the greedy fill algorithm, then ENT$(f_G) \leq$ ENT$(f_{\rm OPT}) + 3$.

Experiments were performed using the Greedy Fill Algorithm to calculate the limit on test data compression for the dynamically compacted test cubes generated by MINTEST [22] for the largest ISCAS'89 [23] benchmark circuits. These are the same test sets used for experiments in [14]–[16], [10], and [24]. Table V shows the maximum percentage compression values for the particular test cubes for each benchmark circuit for four different symbol lengths. The compression values in Table V are calculated from the values of entropy that were generated using the greedy fill algorithm. No fixed-to-fixed or fixed-to-variable code using these particular symbol lengths can achieve greater compression than the bounds shown in Table V (within the approximate factor) for these particular test sets. Note, however, that these entropy bounds would be different for a different test set for these circuits, e.g., if the method in [25] was used to change the location or number of don't cares. However, given any test set, the proposed method can be used

TABLE VI
ENTROPY COMPRESSION LIMITS USING ALTERNATE FILL ALGORITHM

Circuit	Symbol Length									
	4	6	8	10	12	16	20	24	28	32
s5378	52.0 %	56.2 %	59.2 %	63.7 %	66.2 %	71.6 %	76.6 %	85.2 %	91.1 %	96.0 %
s9234	53.6 %	56.9 %	60.0 %	63.4 %	65.0 %	68.7 %	71.8 %	82.4 %	88.5 %	91.8 %
s13207	83.6 %	85.0 %	85.6 %	87.0 %	88.0 %	88.8 %	89.2 %	93.2 %	96.6 %	98.3 %
s15850	68.9 %	70.5 %	71.9 %	73.3 %	75.1 %	77.8 %	80.1 %	85.1 %	89.1 %	94.0 %
s38417	60.1 %	63.2 %	65.3 %	67.7 %	68.5 %	71.7 %	76.6 %	83.0 %	87.7 %	89.2 %
s38584	65.9 %	67.6 %	68.8 %	70.8 %	72.2 %	75.1 %	77.4 %	84.7 %	88.1 %	93.7 %

to determine the corresponding entropy for it. As can be seen from the table, the percentage compression that can be achieved increases with an increase in the symbol length.

The greedy fill algorithm is exponential in the symbol length since the algorithm enumerates the frequencies of each possible minterm for the given symbol length and the number of minterms grows exponentially with symbol length. Thus, it is not practical to calculate the entropy for larger symbol lengths using this algorithm. Hence, the next section presents a polynomial time approximate algorithm that can scale to larger symbol lengths.

2) Alternate Fill Algorithm: Algorithm 2 shows the pseudocode for the alternate fill algorithm. First, the three-valued unspecified symbols are ordered from the highest frequency of occurrence to the lowest. Then, each unspecified symbol is used as the starting point, and an attempt is made to merge it (similar to static compaction) with each of the other symbols going in order from the highest frequency to the lowest. Two symbols can be merged if they do not conflict in any bit position (i.e., have opposite specified values in a bit position). After merging with as many symbols as possible, the resulting merged symbol will have a frequency of occurrence equal to the sum of the frequency of occurrence for each of the symbols it was merged with. Each symbol is used in turn as a starting point for this merging process, and the merged symbol that results in the highest frequency of occurrence is selected (if it has any don't cares after merging, they can be arbitrarily filled with no impact on entropy). The don't cares in each unspecified symbol that are compatible with the selected merged symbol are specified to match it. This process is then repeated until all the symbols in the test data are specified.

```
Algorithm 2: Alternate Fill Algorithm
Input: Test Set $T$ (with unspecified bits), Symbol Length $n$
Output: Test Set $T^{\#}$
APPROX FILL$(T, n)$
  $T^{\#} = T$;
  $S = \{s : s$ is a symbol of size $n$ occurring in $T\}$;
  frequency$(S)$ = CALCULATE_SYMBOL_
  FREQUENCY$(S, T)$;
  while ($S$ not empty)
    SORT$(S$, frequency$(S))$;
    foreach (symbol $s$ of $S$)
      curr_symbol $c = s$;
      curr_freq $= 0$;
      foreach (symbol $s$ of $S$)
        if ($s$ merges with $c$)
          curr_freq$+=$ frequency$[s]$;
          $c =$ MERGE$(s, c)$;
        endif
      done
      if(curr_freq $>$ max_freq)
        max_freq = curr_freq;
        max_freq_symbol $s^* = c$;
      endif
    done
    FULLY_SPECIFY$(s^*)$;
    foreach (symbol $s$ of $S$)
      if ($s$ merges with $s^*$)
        replace $s$ with $s^*$ in $T^{\#}$;
        $S = S - \{s\}$;
      endif
    done
    $S = S - \{s^*\}$;
  endwhile
END
```

The alternate algorithm is approximate because the sequence in which the merging is done may not be optimal. Since the number of sequences in which the merging can be done is exponential, a greedy heuristic is used with some initial lookahead. The alternate algorithm is polynomial time with respect to the number of symbols and symbol length. Hence, it is scalable to large test sets and large symbol lengths.

Experiments were performed using the alternate fill algorithm to calculate the limit on test data compression for the dynamically compacted test cubes generated by MINTEST [22] for the largest ISCAS'89 [23] benchmark circuits. The results are shown in Table VI. In comparing the results in Tables V and VI, it can be seen that the results for the alternate fill algorithm are very close to those of the greedy fill algorithm.

III. Symbol Length Versus Compression Limits

In this section, the relationship between symbol length, compression limits, and decoding complexity is investigated. Fig. 2 shows a graph of the compression limits calculated from minimum entropy (using the alternate algorithm) for the benchmark circuit *s9234* as the symbol length is varied. The percentage compression varies from 50% for a symbol length of 2 to 92% for a symbol length of 32; this corresponds to a compression ratio range of 2–12. The reason why greater compression can be achieved for larger symbol lengths is that more information is being stored in the decoder. This becomes very clear when the

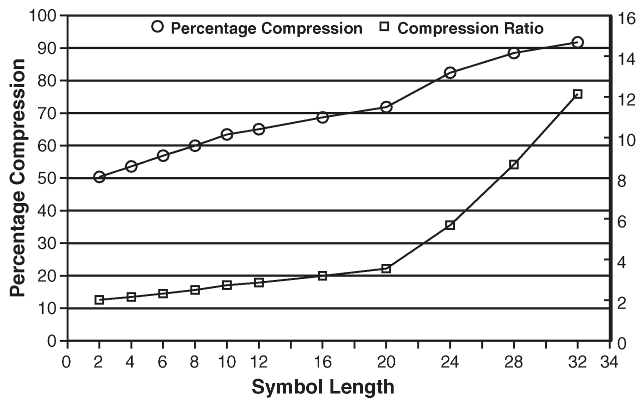

Fig. 2. Variation of compression for *s9234* for different symbol lengths.

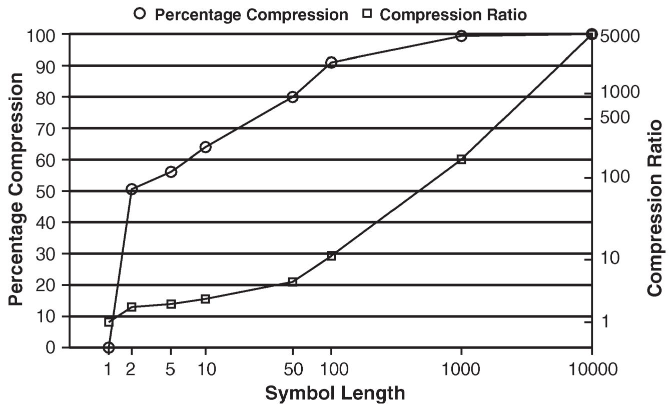

Fig. 3. Plot of compression for *s9234* with all possible symbol lengths.

compression limit is graphed for all possible symbol lengths as shown in Fig. 3. As can be seen, the compression ratio goes toward infinity as the symbol length becomes equal to the number of bits in the entire test set. When the symbol length is equal to the number of bits in the entire test set, then the decoder is encoding the entire test set. This is equivalent to built-in self-test (BIST) where no data are stored on the tester other than the start signal for BIST.

Consider the simple case where the information in the decoder is simply stored in a ROM without any encoding. The decoder ROM translates the codewords into the original symbols; thus, at a minimum, it needs to store the set of original symbols. For a symbol length of 2, there are 2^2 possible symbols; therefore, the decoder ROM would have to store four symbols each requiring 2 bits, thus requiring 8 bits of storage. For a symbol length of 4, there are 2^4 possible symbols; therefore, the decoder ROM would have to store 16 symbols each requiring 4 bits, thus requiring 64 bits of storage. Having a larger decoder ROM allows greater compression. For a symbol length of 32, there are 2^{32} possible symbols, but it is not likely that the test data would contain all of them. The decoder would only need to store the symbols that actually occur in the test data. In Fig. 4, the size of a simple decoder ROM that is required for different symbol lengths is graphed for different symbol lengths. As can be seen, the decoder information goes up exponentially as the symbol length goes from 1 to 10 as, in this range, all possible symbols are occurring in the test data and the number of symbols is growing exponentially. After this, the decoder information goes up less than exponentially because not all possible symbols of that length occur in the test data. After the symbol length exceeds about 20, the decoder information is nearly equal to the entire test set as there is very little repetition of the symbols in the test data (i.e., almost all the symbols in the test data are unique at that point). One way to view the graph in Fig. 4 is that as the symbol length is increased, essentially more information is being stored in the on-chip decoder, and less information is being stored off-chip in the tester memory. A symbol length of 1 corresponds to conventional external testing with no compression, and a symbol length equal to the size of the test set corresponds to conventional BIST where no data are stored on the tester.

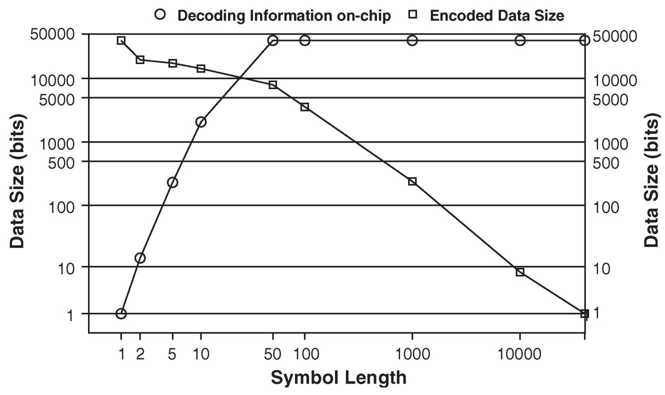

Fig. 4. Encoded data size versus simple decoder ROM size for *s9234*.

The real challenge in test data compression is in the design of the decoder. The entropy of the test data places a fundamental limit on the amount of compression that can be achieved for a particular symbol length, but the real key is to achieve something close to the maximum compression using a small decoder. The next two sections evaluate existing test data compression schemes with respect to their compression limits based on entropy theory.

IV. Analysis of Test Data Compression Schemes

In this section, different test data compression schemes proposed in the literature are compared with their entropy bounds. Both schemes that operate on fixed-length symbols and those that work on variable-length symbols are considered. One major constraint is that we are only able to report results for schemes for which we have access to the exact test set that was encoded. For this reason, we have limited the results to only those schemes that reported results for the MINTEST test cubes [22]. Discussion of LFSR reseeding schemes is deferred to Section V.

A. *Fixed-Symbol-Length Schemes*

Table VII compares results of four compression schemes with the calculated entropy limits for compression. Two of them, i.e., [10] and [24], are fixed-to-variable schemes; the work in [10] is a selective Huffman coding scheme, whereas the work in [24] uses only nine codewords to encode the whole test set. The other two are dictionary-based fixed-to-fixed schemes; the work in [7] uses a selective dictionary with

TABLE VII
COMPARISON OF FIXED-SYMBOL-LENGTH SCHEMES WITH ENTROPY COMPRESSION LIMIT

Circuit	Test Set Size	Symbol Length	Compression Limit	[10]	[24]	[7]	[26]
s5378	23754	8	**59.2 %**	50.1 %	50.7 %	-	-
		12	**66.2 %**	55.1 %	51.6 %	-	-
		16	**71.6 %**	-	49.4 %	45.2 %	28.0 %
		32	**81.8 %**	-	39.4 %	67.2 %	57.5 %
		64	**88.9 %**	-	-	73.3 %	77.6 %
s9234	39273	8	**60.0 %**	50.3 %	50.9 %	-	-
		12	**65.0 %**	54.2 %	46.5 %	-	-
		16	**68.7 %**	-	41.9 %	46.7 %	23.1 %
		32	**80.3 %**	-	26.4 %	65.9 %	57.9 %
		64	**88.9 %**	-	-	67.4 %	77.3 %
		128	**94.2 %**	-	-	70.7 %	88.7 %
s13207	165200	8	**85.6 %**	69.0 %	79.8 %	-	-
		12	**88.0 %**	77.0 %	81.8 %	-	-
		16	**88.8 %**	-	82.3 %	49.2 %	24.6 %
		32	**92.3 %**	-	79.0 %	73.4 %	59.1 %
		64	**94.9 %**	-	-	85.4 %	79.6 %
		128	**96.7 %**	-	-	91.5 %	88.0 %
s15850	76986	8	**71.9 %**	60.0 %	66.4 %	-	-
		12	**75.1 %**	66.0 %	65.1 %	-	-
		16	**77.8 %**	-	63.0 %	47.4 %	23.4 %
		32	**85.2 %**	-	55.1 %	67.6 %	57.4 %
		64	**91.3 %**	-	-	75.9 %	77.1 %
		128	**95.3 %**	-	-	82.0 %	87.7 %
s38417	164736	8	**65.3 %**	55.1 %	60.6 %	-	-
		12	**68.5 %**	59.0 %	59.4 %	-	-
		16	**71.7 %**	-	57.5 %	44.0 %	12.5 %
		32	**80.0 %**	-	47.4 %	49.0 %	53.1 %
		64	**87.6 %**	-	-	42.7 %	75.0 %
		128	**93.4 %**	-	-	36.8 %	86.7 %
s38584	199104	8	**68.8 %**	58.3 %	65.5 %	-	-
		12	**72.2 %**	64.1 %	64.4 %	-	-
		16	**75.1 %**	-	62.4 %	45.8 %	12.0 %
		32	**82.9 %**	-	52.1 %	64.3 %	52.9 %
		64	**89.6 %**	-	-	70.9 %	74.9 %
		128	**94.2 %**	-	-	70.8 %	86.9 %

fixed-length indexes, whereas the work in [26] uses a correction circuit in addition to the dictionary. The first two columns in Table VII show the circuit name and the size of the original test set. The next two columns show the symbol length and the corresponding entropy compression limit (i.e., maximum possible compression). Several different symbol lengths are considered (rows 8, 12, 16, 32, 64, and 128) and the maximum possible compression that can be obtained from entropy limits calculated for them. The last four columns show the published results for the percentage compression obtained by each of the aforementioned schemes.

It is clear from Table VII that different schemes work with different symbol lengths. Selective Huffman [10] and nine-coded [24] compressions are primarily applicable with smaller symbol lengths. The reason for [10] is that the decompressor size increases for bigger symbol lengths. Nine-coded compression [24] performs well for small symbol lengths, but the compression decreases significantly as the symbol length is increased. Since it fits every n-bit block into one of the nine categories, with larger n, most of the blocks do not have a match with the standard categories (all zeroes, all ones) and fall into the all mismatch category, thereby reducing the compression.

The two dictionary-based methods, i.e., [7] and [26], are applicable with larger symbol lengths. For larger symbol lengths, the difference between the entropy compression limits and the actual compression obtained by these schemes is much higher than for smaller symbol lengths. This can be attributed to the fact that these schemes use a simple approach (so that the hardware overhead does not increase exponentially), while to get compression similar to the entropy compression limit, the hardware overhead may be prohibitively high.

TABLE VIII
COMPARISON OF VARIABLE-SYMBOL-LENGTH SCHEMES WITH ENTROPY COMPRESSION LIMIT

Circuit	Test Data Size	Compr. Limit	Golomb [14]	FDR [15]	VIHC [16]
s5378	23754	52.4 %	40.7 %	48.0 %	51.8 %
s9234	39723	47.8 %	43.3 %	43.6 %	47.2 %
s13207	165200	83.7 %	74.8 %	81.3 %	83.5 %
s15850	76986	68.2 %	47.1 %	66.2 %	67.9 %
s38417	164736	54.5 %	44.1 %	43.3 %	53.4 %
s38584	199104	62.5 %	47.7 %	60.9 %	62.3 %

B. Variable-Symbol-Length Schemes

Table VIII shows results for three compression schemes that are based on variable-to-variable codes. Each of these schemes is based on encoding runs of zeroes. As was discussed in Section II, filling the don't cares to minimize entropy for variable size symbols is an open problem. These three schemes all fill the don't cares by simply replacing them with specified zeroes. For simplicity, the entropy compression limits in Table VIII were calculated by also filling the don't cares with specified zeroes.

The results in Table VIII indicate that the VIHC method in [16] gets very close to the entropy compression limit. This is because this method is based on Huffman coding. One conclusion that can be drawn from these results is that there is not much room for improvement for research in variable-to-variable codes that encode runs of zeroes. The only way to get better compression than the entropy compression limit that is shown here would be to use a different ATPG procedure or fill the don't cares in a different way that changes the entropy compression limit.

V. Analysis of LFSR Reseeding

Conventional LFSR reseeding [3] is a special type of fixed-to-fixed code in which the set of symbols is the output space of the LFSR and the set of codewords is the seed of the LFSR. As was seen in the graphs in Section III, larger symbol lengths are needed to achieve greater amounts of compression due to the entropy limits on compression. The difficulty with larger symbol lengths is that the decoder needs to be able to produce more symbols of greater length. The power of LFSR reseeding is that an LFSR is used for producing the symbols. An r-stage LFSR (implementing a primitive polynomial) has a maximal output space as it produces $2^r - 1$ different symbols as well as have a very compact structure resulting in low area. Thus, an LFSR is an excellent vehicle for facilitating larger symbol lengths with a small area decoder. The only issue is whether the set of symbols that occur in the test data, S_{data}, are a subset of the symbols produced by the LFSR, S_{LFSR} (i.e., $S_{\mathrm{data}} \subseteq S_{\mathrm{LFSR}}$). Fortunately, the symbols produced by the LFSR have a pseudorandom property. If n is the symbol length, then as was shown in [3], if the number of specified bits in any n-bit block of test data is 20 less than the size of the LFSR, then the probability of the n-bit block of test data not matching one of the symbols in S_{LFSR} is negligibly small (less than 10^{-6}). In [27], a multiple-polynomial technique was presented that reduces the size of the seed information to just 1 bit more than the maximum number of specified bits (denoted by $s_{\max}$) in an n-bit block. Thus, the compression that is achieved with this approach for a symbol length of n is equal to the ratio of n to the maximum number of specified bits in an n-bit block plus 1. Note that although the compressed data size for LFSR reseeding depends on $s_{\max}$, the hardware overhead in implementing the scheme need not be that high. Several techniques have been proposed that reduce the size of the LFSR by either using the existing scan chains in the circuit like [28] or using a smaller LFSR but continuously feeding it bits from the tester [29].

Fig. 5 shows a graph of the compression for LFSR reseeding for circuit *s13207* for all possible symbol lengths. As can be seen, no compression is achieved for short symbol lengths where the maximum number of specified bits is equal to the symbol length. Compression does not start to occur until the symbol length becomes larger than 20. The compression steadily improves as the symbol length is increased, but in general, it cannot exceed the total percentage of don't cares in the test data. For the test set in Fig. 5, the percentage of don't cares is 94%.

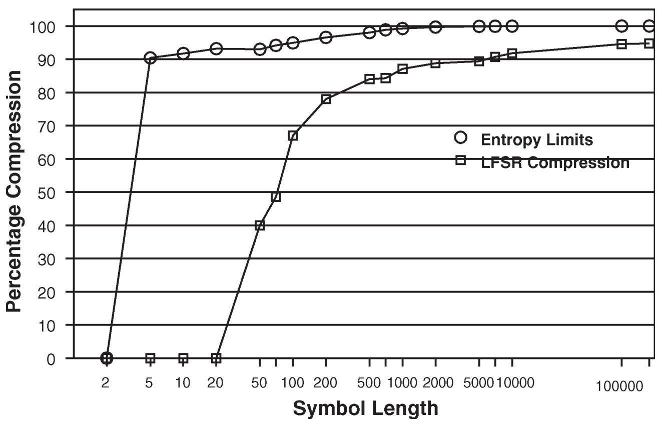

Fig. 5. LFSR reseeding compression versus symbol length for *s13207*.

In another experiment, we evaluated the performance of LFSR reseeding vis-a-vis other test-encoding-based compression schemes. The idea was to compare the compression obtained using LFSR reseeding with the entropy limits of the other compression techniques. The experimental setup is shown in Fig. 6. A scan architecture with n scan chains and a maximum scan length of l was considered. The test-encoding-based compression techniques generate bits corresponding to one scan slice at a time. This is shown by the shaded rectangle on the left side of Fig. 6. That is, the symbol length is n. On the other hand, the LFSR reseeding scheme generates one full test vector from a single seed. Note that the size of the LFSR will be determined by the $s_{\max}$ of the test set and may be different from n. In this case, $s_{\max}$ is the maximum number of specified bits among all test vectors in the test set. Test sets were generated randomly for different scan architectures by varying the number of scan chains (n) and also with several different percentages of specified bits for each scan architecture. For each of these test sets, the entropy limits for the code-based decompression technique were calculated. The compression obtained through LFSR reseeding was evaluated for comparison. The size of the LFSR was assumed to be $s_{\max} + 1$.

Fig. 7 shows the plot of percentage compression versus percentage specified bits. The percentage specified bits in each test set was varied from 0.1% to 5%. Note that this is the overall average percentage specified bits for the whole test set. The number of specified bits in each test pattern varied since they were specified randomly. Since the LFSR reseeding compression does not depend on n, the plot of LFSR reseeding for the three different n in this experiment is identical. From Fig. 7, it is clear that the entropy limits for the test-encoding-based techniques do not vary much with the number of scan chains n for the same percentage specified bits. However, the entropy limits decrease with the increase in the percentage of specified bits. Similarly, the compression obtained using LFSR reseeding also decreases with the increase in percentage of specified bits. However, it is interesting to note that the compression obtained through LFSR reseeding can match that of the entropy limit for lower percentage specified bits and is

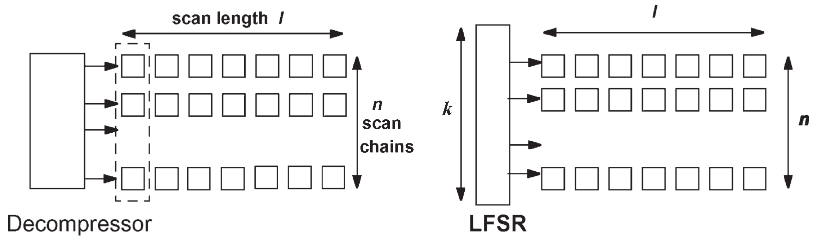

Fig. 6. Experimental setup.

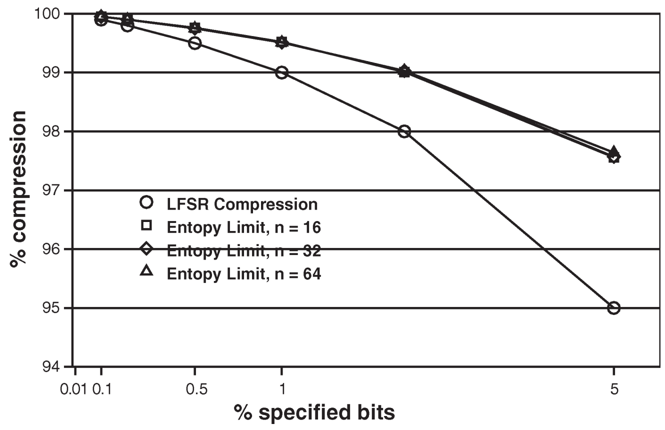

Fig. 7. LFSR compression.

within 3% of the limit for higher percentage specified bits. This means that LFSR reseeding can achieve compression very close to the entropy limits of test-encoding-based techniques.

VI. Conclusion

This paper shows how entropy theory can be used to calculate theoretical limits to the amount of test data compression that can be achieved with various fixed-symbol-length coding techniques. These limits are useful as a measure of how much improvement is possible over existing test data compression schemes. They can be used to identify coding techniques where there is not much scope for improvement as well as identify coding techniques that hold promise for fruitful research. This paper studied the relationship of symbol partitioning and don't care filling to entropy. An area for future research is to look at new coding techniques that can exploit this to achieve greater compression.

References

[1] E. Halperin and R. M. Karp, "The minimum-entropy set cover problem," *Theor. Comput. Sci.*, vol. 348, no. 2, pp. 240–250, Dec. 2005.
[2] Y. Zorian, E. J. Marinissen, and S. Dey, "Testing embedded core based system chips," in *Proc. IEEE Int. Test Conf.*, 1998, pp. 130–143.
[3] B. Könemann, "LFSR-coded test patterns for scan designs," in *Proc. Eur. Des. Test Conf.*, 1991, pp. 237–242.
[4] I. Hamzaoglu and J. H. Patel, "Reducing test application time for full scan embedded cores," in *Proc. Int. Symp. Fault Tolerant Comput.*, 1999, pp. 260–267.
[5] I. Bayraktaroglu and A. Orailoglu, "Test volume and application time reduction through scan chain concealment," in *Proc. Des. Autom. Conf.*, 2001, pp. 151–155.
[6] S. Reddy, K. Miyase, S. Kajihara, and I. Pomeranz, "On test data volume reduction for multiple scan chain designs," in *Proc. IEEE VLSI Test Symp.*, 2002, pp. 103–108.
[7] L. Li, K. Charabarty, and N. A. Touba, "Test data compression using dictionaries with selective entries and fixed-length indices," *ACM Trans. Des. Automat. Electron. Syst.*, vol. 8, no. 4, pp. 470–490, Oct. 2003.
[8] D. A. Huffman, "A method for the construction of minimum redundancy codes," *Proc. IRE*, vol. 40, no. 9, pp. 1098–1101, Sep. 1952.
[9] A. Jas, J. Dastidar, and N. Touba, "Scan vector compression/decompression using statistical coding," in *Proc. VLSI Test Symp.*, 1999, pp. 114–120.
[10] A. Jas, J. G. Dastidar, M.-E. Ng, and N. Touba, "An efficient test vector compression scheme using selective Huffman coding," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 22, no. 6, pp. 797–806, Jun. 2003.
[11] A. Jas and N. Touba, "Test vector decompression via cyclical scan chains and its application to testing core based designs," in *Proc. Int. Test Conf.*, 1998, pp. 458–464.
[12] F. Wolff and C. Papachristou, "Multiscan-based test compression and hardware decompression using LZ77," in *Proc. IEEE Int. Test Conf.*, 2002, pp. 331–338.
[13] M. J. Knieser *et al.*, "A technique for high ratio LZW compression," in *Proc. Des. Autom. Test Eur.*, 2003, pp. 116–121.
[14] A. Chandra and K. Chakrabarty, "System-on-a-chip test data compression and decompression architectures based on Golomb codes," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 20, no. 3, pp. 355–368, Mar. 2001.
[15] ——, "Frequency-directed run length (FDR) codes with application to system-on-a-chip test data compression," in *Proc. VLSI Test Symp.*, 2001, pp. 42–47.
[16] P. T. Gonciari, B. M. Al-Hashimi, and N. Nicolici, "Variable-length input Huffman coding for system-on-a-chip test," *IEEE Trans. Comput.-Aided Design Integr. Circuit Syst.*, vol. 22, no. 6, pp. 783–796, Jun. 2003.
[17] A. Chandra and K. Chakrabarty, "How effective are compression codes for reducing test data volume," in *Proc. VLSI Testing Symp.*, 2002, pp. 91–96.
[18] K. J. Balakrishnan and N. Touba, "Relating entropy theory to test data compression," in *Proc. Eur. Test Symp.*, 2004, pp. 94–99.
[19] T. Cover and J. A. Thomas, *Elements of Information Theory*, 2nd ed. Hoboken, NJ: Wiley, 1992.
[20] A. Wurtenberger, C. Tautermann, and S. Hellebrand, "A hybrid coding strategy for optimized data compression," in *Proc. IEEE Int. Test Conf.*, 2003, pp. 451–459.
[21] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, 2nd ed. Cambridge, MA: MIT Press.
[22] I. Hamzaoglu and J. H. Patel, "Test set compaction algorithms for combinational circuits," in *Proc. ICCAD*, 1998, pp. 283–289.
[23] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *Proc. Int. Symp. Circuits and Syst.*, 1989, pp. 1929–1934.
[24] M. Tehranipour, M. Nourani, and K. Chakrabarty, "Nine-coded compression technique with application to reduced pin-count testing and flexible on-chip decompression," in *Proc. Des., Autom. Test Conf. Eur.*, 2004, pp. 1284–1289.
[25] S. Kajihara, K. Taniguchi, K. Miyase, I. Pomeranz, and S. Reddy, "Test data compression using don't care identification and statistical encoding," in *Proc. Asian Test Symp.*, 2002, pp. 67–72.
[26] A. Wurtenberger, C. Tautermann, and S. Hellebrand, "Data compression for multiple scan chains using dictionaries with corrections," in *Proc. IEEE Int. Test Conf.*, 2004, pp. 926–935.
[27] S. Hellebrand, J. Rajski, S. Tarnick, S. Venkataraman, and B. Courtois, "Built-in test for circuits with scan based on reseeding of multiple-polynomial linear feedback shift registers," *IEEE Trans. Comput.*, vol. 44, no. 2, pp. 223–233, Feb. 1995.
[28] N. Zacharia, J. Rajski, and J. Tyszer, "Decompression of test data using variable-length seed LFSRs," in *Proc. VLSI Test Symp.*, 1995, pp. 426–433.

[29] J. Rajski *et al.*, “Embedded deterministic test for low cost manufacturing test,” in *Proc. Int. Test Conf.*, 2002, pp. 301–310.

[30] K. J. Balakrishnan, “New approaches and limits to test data compression for systems-on-chip,” Ph.D. dissertation, Univ. Texas, Austin, TX, 2004.

Kedarnath J. Balakrishnan (M’00) received the B.Tech. degree in electrical engineering from Indian Institute of Technology, Bombay, India, in 2000, and the M.S. and Ph.D. degrees in electrical and computer engineering from the University of Texas at Austin in 2002 and 2004, respectively.

From 2004 to 2006, he was a Research Staff Member with NEC Laboratories America, Princeton, NJ, leading projects on test data compression and system-on-a-chip testing. He is currently with the DFT/ATPG Group, Advanced Micro Devices, Inc., Austin, TX. His research interests include test data compression, microprocessor testing, and design for test at the RTL level.

Nur A. Touba (SM’05) received the B.S. degree from the University of Minnesota, Minneapolis, in 1990, and the M.S. and Ph.D. degrees from Stanford University, Stanford, CA, in 1991 and 1996, respectively, all in electrical engineering.

He is currently an Associate Professor with the Department of Electrical and Computer Engineering, University of Texas at Austin.

Dr. Touba is on the Program Committee for the International Test Conference, International Conference on Computer Design, Design Automation and Test in Europe Conference, International On-Line Test Symposium, European Test Symposium, Asian Test Symposium, Defect and Fault Tolerance Symposium, Microprocessor Test and Verification Workshop, International Workshop on Open Source Test Technology Tools, and International Test Synthesis Workshop. He received the National Science Foundation Early Faulty CAREER Award in 1997 and the Best Paper Award at the 2001 VLSI Test Symposium.