An Efficient Test Vector Compression Scheme Using Selective Huffman Coding

Abhijit Jas, Jayabrata Ghosh-Dastidar, Mom-Eng Ng, and Nur A. Touba

***Abstract*—This paper presents a compression/decompression scheme based on selective Huffman coding for reducing the amount of test data that must be stored on a tester and transferred to each core in a system-on-a-chip (SOC) during manufacturing test. The test data bandwidth between the tester and the SOC is a bottleneck that can result in long test times when testing complex SOCs that contain many cores. In the proposed scheme, the test vectors for the SOC are stored in compressed form in the tester memory and transferred to the chip where they are decompressed and applied to the cores. A small amount of on-chip circuitry is used to decompress the test vectors. Given the set of test vectors for a core, a modified Huffman code is carefully selected so that it satisfies certain properties. These properties guarantee that the codewords can be decoded by a simple pipelined decoder (placed at the serial input of the core's scan chain) that requires very small area. Results indicate that the proposed scheme can provide test data compression nearly equal to that of an optimum Huffman code with much less area overhead for the decoder.**

***Index Terms*—Automatic test equipment, compression, decompression architecture, embedded core testing, testing time, test set encoding.**

I. Introduction

One of the key concerns in any design project is to meet time-to-market constraints. In order to accomplish this goal, chip designers often use predesigned and preverified cores to develop systems-on-a-chip (SOC) devices. With time, these devices have become extremely complex. This high level of integration has allowed vendors to drive down the effective manufacturing costs. However, it has also rapidly increased the complexity of testing these chips. One of the increasingly difficult challenges in testing SOCs is dealing with the large amount of test data that must be transferred between the tester and the chip [9], [34]. Each core in an SOC has a given set of test vectors that must be applied to it (usually through a test wrapper that is provided around a core). The test vectors must be stored on the tester and then transferred to the inputs of the core during modular testing. As more and more cores (each with its own test set) are placed on a single chip, the amount of total test data for the chip increases rapidly. This poses a serious problem because of the cost and limitations of automated test equipment (ATE). Testers have limited speed, channel capacity, and memory. In general, the amount of time required to test a chip depends on how much test data needs to be transferred to the chip and how fast the data can be transferred (i.e., the test data bandwidth to the chip). This depends on the speed and channel capacity of the tester and the organization and characteristics of the scan chains on the chip. Both test time and test storage are major concerns for SOCs from a test economics point of view.

This paper presents a statistical compression/decompression scheme to reduce the amount of test data that needs to be stored on the tester and transferred to the chip (preliminary results were published in [24]). The idea is to store the test vectors for a core in the tester memory in compressed form, and then transfer the compressed vectors to the chip, where a small amount of on-chip circuitry is used to decompress the test vectors. Instead of having to transfer each entire test vector from the tester to the core, a smaller amount of compressed data is transferred instead. The approach presented here significantly reduces both test storage requirements and the overall test time.

Transferring compressed test vectors takes less time than transferring the full vectors at a given bandwidth. However, in order to guarantee a reduction in the overall test time, the decompression process should not add delay (which would subtract from the time saved in transferring the test data). Moreover, the on-chip decompression circuitry must be small so that it does not add significant area overhead. Given a set of test vectors, a method is presented here for choosing a statistical code (a modified form of Huffman coding) that can be decoded with a simple pipelined decoder. The properties of the code are chosen such that the pipelined decoder has a very small area and is guaranteed to be able to decode the test data as fast as the tester can transfer it.

The compression/decompression scheme presented in this paper can be used for generating any set of deterministic scan vectors. It preserves the sequence of the vectors and requires no modifications to the circuit-under-test (CUT). It does not require any knowledge of the internal design of the CUT, and, thus, is suitable for testing intellectual property cores where the core supplier does not provide any information about the internal structure of the core.

II. Related Work

The problem of reducing test time and test data for core-based SOCs has been attacked from several different angles in recent literature. Novel approaches for compressing test data using the Burrows–Wheeler transform and run-length coding were presented in [21], [33]. These schemes were developed for reducing the time to transfer test data from a workstation across a network to a tester (not for use on chips). Scan chain architectures for core-based designs that maximize bandwidth utilization are presented in [1]. A technique for compression/decompression of scan vectors using cyclical decompressors and run-length coding is described in [23]. A modular built-in self-test (BIST approach that allows sharing of BIST control logic among multiple cores is presented in [30]. A novel technique for combining BIST and external testing across multiple cores is described in [32]. The idea of statistically encoding test data was presented in [22]. They described a BIST scheme for nonscan circuits based on statistical coding using comma codes (very similar to Huffman codes) and run-length coding. An approach called "parallel serial full scan (PSFS) for reducing test time in cores is presented in [16]. A technique to reduce test data and test time by using specially designed cores (cores with virtual scan chains) is presented in [26]. An approach that uses a linear combinational expander circuit is described in [2]. The use of Golomb codes and frequency-directed run-length (FDR) codes for compressing test data have been demonstrated in [5]–[7], respectively. The use of variable length input Huffman

Manuscript received July 6, 2002; revised September 30, 2002. This work was supported in part by the National Science Foundation under Grant no. MIP-9702236 and in part by the Texas Advanced Research Program under Grant no. 1997-003658-369. This paper was recommended by Associate Editor K. Chakrabarty.

A. Jas was with the Department of Electrical and Computer Engineering, University of Texas, Austin, TX 78712-1084. He is now with the Intel Corporation, Austin, TX 78746 USA.

J. Ghosh-Dastidar was with the Department of Electrical and Computer Engineering, University of Texas, Austin, TX 78712-1084 USA. He is now with the Altera Corporation, San Jose, CA 95134 USA.

M.-E. Ng was with the Department of Electrical and Computer Engineering, University of Texas, Austin, TX 78712-1084. She is now with Advanced Micro Devices, Austin, TX 78741 USA.

N. A. Touba is with the Computer Engineering Research Center, Department of Electrical and Computer Engineering, University of Texas, Austin, TX 78712-1084 USA (e-mail: touba@ece.utexas.edu).

Digital Object Identifier 10.1109/TCAD.2003.811452

0278-0070/03$17.00 © 2003 IEEE

codes for SOC test data compression has been proposed in [14]. A fixed-to-fixed block encoding scheme is described in [28]. Techniques for reusing scan chains from other cores in an SOC to increase the test data bandwidth has been described in [11], and automatic test pattern generation (ATPG) techniques for producing test cubes that are suitable for encoding, using the above technique, have been described in [12]. A fault simulation-based technique to reduce the entropy of the test vector set by pattern transformation is described in [19]. Such transformations increase the amount of compression that can be achieved on the transformed test set using statistical coding. ATPG algorithms for producing test vectors that can more effectively be compressed using statistical codes have been described in [20]. Test vector compression based on hybrid BIST techniques have been described in [10], [29], and [27]. A novel scheme of test vector compression using an embedded processor is described in [25]. In [13], a test vector compression technique based on geometric primitives is proposed. Very recently, a new line of research has focused on compressing test data volume while optimizing other factors like test power [8], [31].

Although all the techniques mentioned above have the same high-level objective (that of reducing test data and/or test application time), the different approaches present different design alternatives and their applicability to a particular design situation varies from case to case. This paper presents a selective Huffman coding scheme for testing cores with internal scan. One of the features of this approach is that the code that is used for a particular core is carefully chosen such that only a small decoder circuit is required. There are no restrictions on the order of the test set, and no modifications need to be made to the core-under-test. The small decoder circuit is simply placed at the serial input of the core's scan chain. As will be shown, the decoder provides a significant reduction in the amount of test data that must be transported from the tester to the core.

III. Statistical Coding

The compression/decompression scheme described in this paper is based on statistical coding. In statistical coding, variable length codewords are used to represent fixed-length blocks of bits in a data set. For example, if a data set is divided into four-bit blocks, then there are 2^4 or 16 unique four-bit blocks. Each of the 16 possible four-bit blocks can be represented by a binary codeword. The size of each codeword is variable (it need not be four bits). The idea is to make the codewords that occur most frequently have a smaller number of bits, and those that occur least frequently to have a larger number of bits. This minimizes the average length of a codeword. The goal is to obtain a coded representation of the original data set that has the smallest number of bits.

A Huffman code [18] is an optimal statistical code that is proven to provide the shortest average codeword length among all uniquely decodable variable length codes. A Huffman code is obtained by constructing a Huffman tree. The path from the root to each leaf gives the codeword for the binary string corresponding to the leaf. An example of constructing a Huffman code can be seen in Table I and Figs. 1 and 2. An example of a test set divided into four-bit blocks is shown in Fig. 1. Table I shows the frequency of occurrence of each of the possible blocks (referred to as symbols). There are a total of 60 four-bit blocks in the example in Fig. 1. Fig. 2 shows the Huffman tree for this frequency distribution and the corresponding codewords are shown in Table I.

An important property of Huffman codes is that they are prefix-free. No codeword is a prefix of another codeword. This greatly simplifies the decoding process. The decoder can instantaneously recognize the end of a codeword uniquely without any look ahead.

TABLE I
Statistical Coding Based on Symbol Frequencies for Test Set in Fig. 1

Symbol	Freq.	Pattern	Huffman Code	Selected Code
S_0	22	0010	10	**10**
S_1	13	0100	00	**110**
S_2	7	0110	110	**111**
S_3	5	0111	010	**0**0111
S_4	3	0000	0110	**0**0000
S_5	2	1000	0111	**0**1000
S_6	2	0101	11100	**0**0101
S_7	1	1011	111010	**0**1011
S_8	1	1100	111011	**0**1100
S_9	1	0001	111100	**0**0001
S_{10}	1	1101	111101	**0**1101
S_{11}	1	1111	111110	**0**1111
S_{12}	1	0011	111111	**0**0011
S_{13}	0	1110	-	-
S_{14}	0	1010	-	-
S_{15}	0	1001	-	-

0010 0100 0010 0110 0000 0010 1011 0100 0010 0100 0110 0010
0010 0100 0010 0110 0000 0110 0010 0100 0110 0010 0010 0000
0010 0110 0010 0010 0010 0100 0100 0110 0010 0010 1000 0101
0001 0100 0010 0111 0010 0010 0111 0111 0100 0100 1000 0101
1100 0100 0100 0111 0010 0010 0111 1101 0010 0100 1111 0011

Fig. 1. Example of test set divided into four-bit blocks.

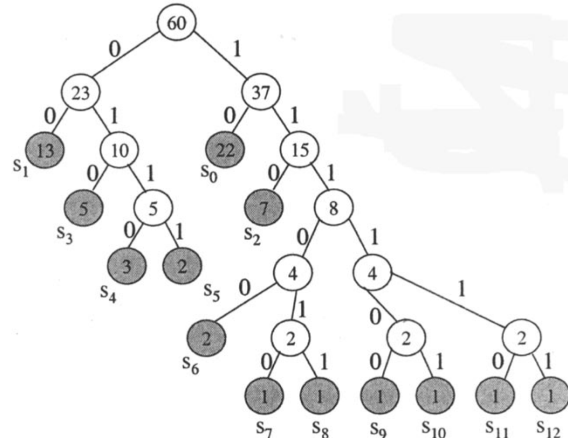

Fig. 2. Huffman tree for the code shown in Table I.

The amount of compression that can be achieved with statistical coding depends on how skewed the frequency of occurrence is for the different codewords. If all of the codewords occur with equal frequency, then no compression can be achieved. It is well known, however, that the test vectors in a test set tend to have a lot of correlations. This arises from the fact that faults in the CUT that are structurally related require similar input value assignments in order to be provoked and sensitized to an output. This often results in skewed frequency of occurrence for different codewords. Moreover, for test cubes, the compression can be very large. The don't care bits (X's) provide flexibility to allow a block to be encoded with more than one possible codeword. The shortest possible codeword can be chosen for each block to maximize the compression. Algorithms for filling test cubes for maximizing compression are described in Section VI.

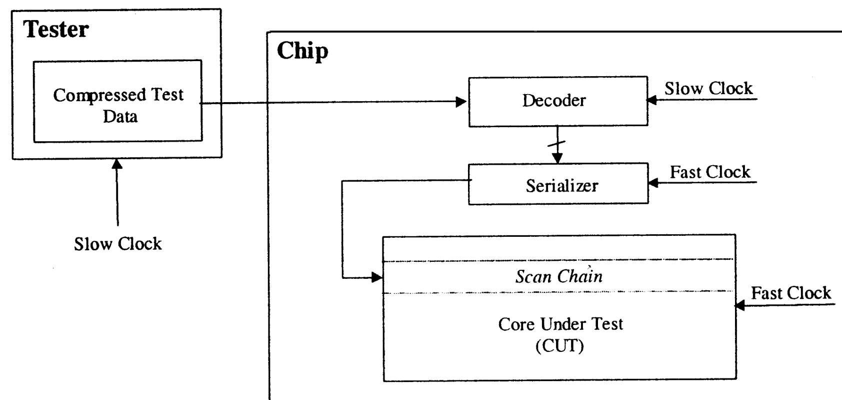

Fig. 3. Block diagram illustrating compression/decompression scheme for a slower tester clock.

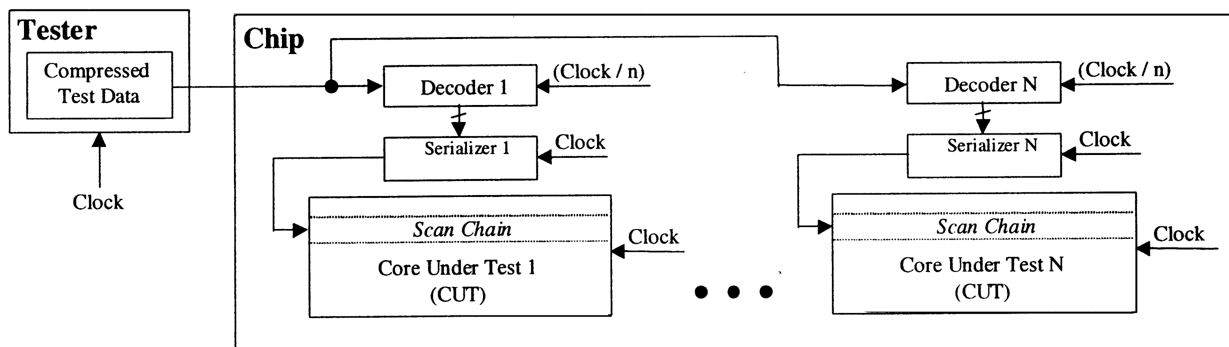

Fig. 4. Block diagram illustrating compression/decompression scheme using single tester channel to feed multiple scan chains.

To fully exploit the correlations in a test set, the number of bits in each scan vector should be a multiple of the fixed-length block size used for the statistical code. When dividing the test set into b-bit blocks for coding, if the size of the scan vectors is not a multiple of b, then X's can be added to pad the start of the vectors (first bits shifted into the scan chain) to make the length a multiple of b. Shifting some extra bits (at the start of the vector) into the scan chain does not matter provided the final contents of the scan chain form the correct test vector when it is applied to the core-under-test. Having each scan vector be a multiple of the block size aligns the blocks within the vectors so that the correlations between the bits will skew the frequencies.

IV. Overview of the Proposed Scheme

The hardware architecture for the proposed scheme is explained in this section for a single scan chain. The compression/decompression scheme proposed here involves statistically coding the scan vectors and then placing an on-chip decoder at the serial input of the scan chain to decompress the vectors. A block diagram illustrating the scheme is shown in Fig. 3. The tester channel shifts a constant stream of variable length codewords (corresponding to compressed scan data) to the decoder. The decoder generates the corresponding fixed-length blocks. At every tester clock cycle the decoder receives one bit from the tester. It takes the decoder L clock cycles to decode a codeword, where L is the length of the codeword. Once the decoder has decoded the codeword, it has to shift the decoded source data into the scan chain. It is not desirable for the tester to wait for the decoder to finish shifting the decoded output into the scan chain. This is because any such wait time induced by the decoder will reduce the test time reduction that can otherwise be obtained by compressing the test data. For this reason, in this scheme, a serializer is used to provide some degree of parallelism between the two operations, one being the receiving of the input bits from the tester and decoding them, and the other being the shifting of the decoded output into the scan chain. Note that the serializer can provide the necessary parallelism in the shift operation because the decoder produces all the bits of the decoded output in parallel (at the same time). If the serializer can shift the decoded output into the scan chain within the time it takes the decoder to decode the next codeword, then the decoder can immediately load the next decoded output into the serializer and continue with the decoding process without having to stop the tester. Since, in many cases, the number of bits b in the fixed-length decoded block is greater than the number of bits in the codeword, the rate at which data needs to be shifted out of the decoder is higher than the rate at which the data is coming into the decoder. There are two ways to achieve this.

1) *Use scan chain with faster clock than tester clock.* This is illustrated in Fig. 3. If the system clock rate is faster than the tester clock rate, then it may be possible to clock the scan chain at a faster clock rate than the tester's clock rate (as described in [17]). The serializer placed between the decoder and the scan chain is then also clocked at the faster system clock rate. The serializer is loaded in parallel by the decoder (allowing the decoder to generate multiple bits of data in a slower tester clock cycle) and serially shifted out into the scan chain at a faster clock rate. One advantage of this approach is that it can be used to provide at-speed scan with a slow tester [17].
2) *Use single tester channel to feed multiple scan chains.* This is illustrated in Fig. 4. If it is not possible to clock the scan chain with a faster clock than the tester clock, then another approach is to have the tester channel rotate between n scan chains (each scan chain has its own decoder). At each clock cycle, the tester shifts

in a bit for a different decoder for each of the n scan chains. Each of the n decoders simply samples its input once every n clock cycles in a different phase from the other decoders. For example, if there are two scan chains $(n = 2)$, then the decoder for scan chain 1 would sample its input on even tester clock cycles, and the decoder for scan chain 2 would sample its input on odd tester clock cycles. With this approach, the "effective clock rate" for each of the decoders is divided by n. However, the scan chain corresponding to each decoder is still clocked at the normal tester clock rate and, thus, its clock rate is n times faster than the decoder. Each time the decoder is clocked once, the scan chain is clocked n times.

In the remainder of this paper, without loss of generality, it will be assumed that the scan clock is faster than the tester clock (i.e., corresponding to scenario 1 above). However, all of the concepts apply equally as well for scenario 2 where the tester channel feeds multiple scan chains such that the "effective clock rate" seen by each decoder is slower than the clock rate of the scan chain.

To illustrate how the decoder and serializer work, consider the following example. Suppose the scan vectors are divided into four-bit blocks, and each four-bit block is replaced by a variable length codeword. The compressed test data stored on the tester consists of the variable length codewords. These codewords are shifted into the decoder as a continuous stream of bits. If the codewords are prefix-free, than the decoder can easily recognize when it has received a complete codeword. When the decoder has received a complete codeword, it loads the corresponding four-bit block in parallel into the serializer. The contents of the serializer are then shifted into the scan chain. If the scan chain is clocked at twice the clock rate that the tester operates at, then after two tester clock periods the entire contents of the serializer will be shifted into the scan chain. During the two tester clock periods that the serializer is in operation, the decoder can be receiving the next codeword.

The key to making the scheme work is careful selection of the statistical code that is used for compressing the test set. There are two important issues that must be considered in selecting the code: one is that the decoder must be small in order to keep the area overhead down, and the other is that the decoder must not output the decompressed bits into the serializer faster than they can be shifted out into the scan chain. While a Huffman code gives the optimum compression for a test set divided into a particular fixed-length block size, it generally requires a very large decoder. A Huffman code for a fixed-length block size of b bits requires a finite state machine (FSM) decoder with $2^b - 1$ states. Thus, the size of the decoder for a Huffman code grows exponentially as the block size is increased. A method for selecting an efficient statistical code for the proposed scheme is described in the following section.

In this scheme, the output response is assumed to be fully compacted on-chip using standard response compaction hardware structures such as a multiple-input signature register (MISR). Test response compaction is an extensively researched topic and several well-defined techniques exist for doing so [4].

V. Statistical Code Selection for Proposed Scheme

Given the test set for a core, a statistical code for compressing the test set must be selected. There is a tradeoff in selecting the code between the amount of compression that is achieved and the complexity of the decoder. Moreover, if the clock frequency of the tester is f_{T} and the clock frequency of the scan chain is f_{sys} (system clock frequency) then the ratio of the system clock frequency and the tester clock frequency $f_{\mathrm{sys}}/f_{\mathrm{T}}$ limits the minimum size of a codeword. If the test set is divided into fixed-length blocks of b bits, then the serializer will hold b bits, and, thus, it takes b scan-clock cycles to shift the buffer's contents into the scan chain. During the time that the contents of the serializer are being shifted into the scan chain, the tester is shifting bits into the decoder. When the decoder receives a complete codeword, it needs to output the corresponding block of b bits into the serializer. If the codeword is too short, then the serializer may not have been emptied yet which would cause a problem for the decoder. So, in order to ensure that the serializer is always empty when the decoder finishes decoding a codeword, the minimum size of a codeword $L_{\min}$ must be no smaller than the ratio of the tester and scan-clock rates times the size of each block

$$L_{\min} \geq b\frac{f_T}{f_{\mathrm{sys}}}. \tag{i}$$

For example, if the block size is 8 and the scan-clock rate is twice the tester clock rate, then the minimum size of a codeword is 4. Note that if it is not possible to have the scan clock rate be faster then the tester clock rate, then an alternative solution (as previously described) is to make the scan clock rate be twice as fast as the "effective clock rate" as seen by the decoder by simply having the tester channel feed two scan chains so that the rate that the decoder receives data from the tester is half as fast as the rate at which data can be shifted into the scan chain.

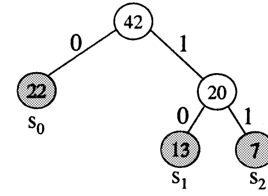

Fig. 5. Huffman tree for the three highest frequency symbols in Table I.

Using a Huffman code would provide the maximum compression, however, it would require a complex decoder and may not satisfy the constraint on the minimum size of a codeword. Therefore, some alternative statistical code must be selected. The approach taken here involves using a selective coding approach for which a very simple decoder can be constructed. Consider the case where the test set is divided into fixed-length blocks of b bits. There will be 2^b codewords. The first bit of each codeword will be used to indicate whether the following bits are coded or not. If the first bit of the codeword is a 0, then the next b bits are not coded and can simply be passed through the decoder as is (hence, the complete codeword has $b + 1$ bits). If the first bit of the codeword is a 1, then the next variable number of bits form a prefix-free code that will be translated by the decoder into a b-bit block. The idea is to only code the most frequently occurring b-bit blocks using codewords with small numbers of bits (less than b, but greater than or equal to $L_{\min}$). Compression is achieved by having the most common b-bit blocks be represented by codewords with less than b bits. The decoder is simple because only a small number of blocks are coded. The vast majority of the blocks are not coded and can be simply passed through the decoder. If n blocks are coded, then the decoder can be implemented with an FSM having no more than $n + b$ states (compared with a Huffman code which requires $2^b - 1$ states).

An example to illustrate the proposed approach for selecting a statistical code is shown in Fig. 5. Consider the test set in Fig. 1. If the entire test set is divided into four-bit blocks then the frequency distribution obtained is shown in the second column of Table I. As can be seen from Table I, the symbols having the highest frequencies are 0010, 0100, and 0110. So, these are the symbols that are coded while the rest of them will be left unchanged. A Huffman tree for the three patterns is constructed to get their codewords (as shown in Fig. 5). The codewords for the remaining 13 symbols are simply a 0 followed by the symbol itself (as shown in the last column of Table I).

The two important parameters in selecting the code are the block size b and the number of coded blocks n. Once those have been chosen, then

```
1 0 X 1 X X 1 0 1 X X X
X 0 1 1 1 0 X 1 1 0 X 1
0 X 1 0 1 0 1 X 1 X X X
```

Fig. 6. Example Test Set to Illustrate Alg1 for Filling Test Cubes.

```
1 0 1 1 0 0 1 0 1 0 1 1
1 0 1 1 1 0 1 1 1 0 1 1
0 0 1 0 1 0 1 1 1 0 1 1
```

Fig. 7. Example test set from Fig. 6 after X's are filled using Alg1.

the procedure for constructing the code is mechanical. A Huffman tree is formed for the n most frequently occurring b-bit blocks. The codewords for the most frequently occurring blocks are simply a 1, followed by the Huffman code obtained from the Huffman tree. The codewords for the remaining blocks are simply a 0, followed by the b-bit block itself. The amount of area overhead for the decoder can be controlled by placing an upper bound on the values of n and b. An increase in n implies an increase in the number of states of the decoder where in the limiting case when all patterns are encoded the decoder becomes a full Huffman decoder. An increase in the block size b, on the other hand, implies an increase in the serializer area that is required for this scheme. In this case, the limit is to make each test vector a pattern which results in a large hardware overhead to regenerate the test vectors from the codewords. The effect of n and b on the amount of compression will be discussed in greater detail in the experimental results section. For a particular value of b, the amount of compression that will be achieved can be computed in linear time with respect to the number of bits in the test set. Thus, the best value of b can be efficiently determined through experimentation. Several values of b can be tried for a particular test set to determine which gives the best compression. Similarly, the best value for n can also be efficiently determined through experimentation.

VI. Algorithms for Filling Test Cubes

One of the advantages of implementing this selective Huffman encoding scheme on test cubes is that the unspecified bits can be filled with 1's and 0's in a way that the frequency distribution of the patterns becomes skewed. This helps in maximizing the compression. There are several algorithms that can be used to fill the X's. In this section, two will be discussed.

When the block size is sufficiently small, an exact analysis can be done by considering all binary combinations (minterms) contained in the unspecified blocks. This algorithm (henceforth, referred to as Alg1) is illustrated with an example in the following paragraphs.

Fig. 6 shows an example test set consisting of three test cubes, each of length 12. Let the block size be $b = 4$. Hence, the three test cubes shown above are partitioned into a set of 9 four-bit blocks, $B = \{10X1,\ XX10,\ 1XXX,\ X011,\ 10X1,\ 10X1,\ 0X10,\ 101X,\ 1XXX\}$. Each unspecified block can contain from 1 (if fully specified) to $2^4 = 16$ (if completely unspecified) possible binary combinations (minterms). For each of the 16 possible minterms for a block, the frequency of occurrence is determined by seeing how many of the unspecified blocks (in set B) contain that minterm. For example, the minterm 1111 is contained in two of the unspecified blocks in the set B, while the minterm 0000 is not contained in any of the unspecified blocks. The minterm that occurs most frequently (i.e., is contained in the largest number of unspecified blocks in set B) is selected first. The X's in each unspecified block that contains the most frequent minterm are specified so that it matches that minterm, and the unspecified block is then removed from the set B. The frequency of occurrence for each of the remaining minterms is then recomputed since the set B has been changed, and the procedure repeats until the set B is empty. This procedure maximizes the frequency of occurrence of the codewords thereby increasing the encoding efficiency of the statistical encoding.

In the example in Fig. 6, the most frequently occurring minterm is 1011. Seven of the unspecified blocks in B contain 1011, so after the first iteration, the set B will contain only $\{XX10$ and $0X10\}$. The most frequently occurring minterm in the second iteration is 0010, which is contained in both remaining unspecified blocks. The test cubes after specifying all the X's is shown in Fig. 7.

Alg1 provides an exact analysis of the frequency distribution of the minterms by considering all possibilities. However, this comes at a cost in terms of the runtime of the algorithm. It is easy to see that the algorithm is exponential in block size b and, hence, the use of this algorithm should be limited to small block sizes only. However, there are alternate ways to specify the don't care bits to maximize the compression which trade off accuracy for faster runtime. The next algorithm (henceforth, referred to as Alg2) is extremely fast and in most cases produces results comparable to the first algorithm. Alg2 is illustrated next with the same example used in the previous case.

In Alg2, the most frequently occurring unspecified block is identified. It is then compared with the next most frequently occurring unspecified block to see if there is a conflict in any bit position (i.e., one has a 1 and the other has a 0, or vise versa). If there is no conflict, then they are merged by specifying all bit positions in which either block has a specified value. For example, if block $X0X1$ is merged with block $X01X$, then the resulting block is $X011$. Note that merging blocks can only increase the number of specified bits. The most frequently occurring unspecified block is compared with all the other unspecified blocks in decreasing order of frequency and whenever merging is possible, it is done. This is done until no more merging can be done with the most frequently occurring unspecified block. This process is then repeated for the second most frequently occurring unspecified block. This continues until there are no more blocks that can be merged. At this point, all the remaining blocks are unique and cannot share any minterms. Any remaining X's can now be randomly filled with 0's and 1's as they will have no impact on the amount of compression. Alg2 fills the X's by greedily merging unspecified blocks based on their frequency of occurrence. This is a heuristic that skews the frequency of occurrence, however, unlike Alg1, it is not guaranteed to maximize the encoding efficiency since the greedy procedure may miss a better merging order. However, it is a much faster procedure than Alg1 as the number of operations is much less because merging is done right away to reduce the set of blocks.

Consider applying Alg2 to the example test data shown in Fig. 6. The set B as described earlier has 6 unique unspecified blocks $10X1$, $XX10$, $1XXX$, $X011$, $0X10$ and $101X$. Let the set of these 6 unique blocks be denoted by B_{uniq}. Of these six unique blocks, the frequency of occurrence of block $10X1$ is 3, that of block $1XXX$ is 2, and, for the rest, the frequency is 1. In the first step of the algorithm, since the block $10X1$ is the most frequently occurring, it is compared with the next most frequently occurring block which is $1XXX$. Since there are no conflicts, they are merged thereby reducing the set B_{uniq}. The merged block $10X1$ is then compared with the other blocks that have frequency 1, and is merged with $X011$ and $101X$. At this point, the set $B_{\mathrm{uniq}} = \{1011,\ XX10,\ 0X10\}$. The procedure is then repeated again, starting with the next most frequently occurring unspecified block. In the end, $B_{\mathrm{uniq}} = \{1011,\ 0X10\}$ and no more merging can be done. The final test vector set is shown in Fig. 8.

Note that unlike the previous algorithm, in this case it is possible to have some don't care bits left over in the transformed test set which can now be randomly filled with 1's or 0's without having any impact on the amount of compression. The amount of compression obtained

```
1 0 1 1 0 X 1 0 1 0 1 1
1 0 1 1 1 0 1 1 1 0 1 1
0 X 1 0 1 0 1 1 1 0 1 1
```

Fig. 8. Example test set from Fig. 6 after X's are filled using Alg2.

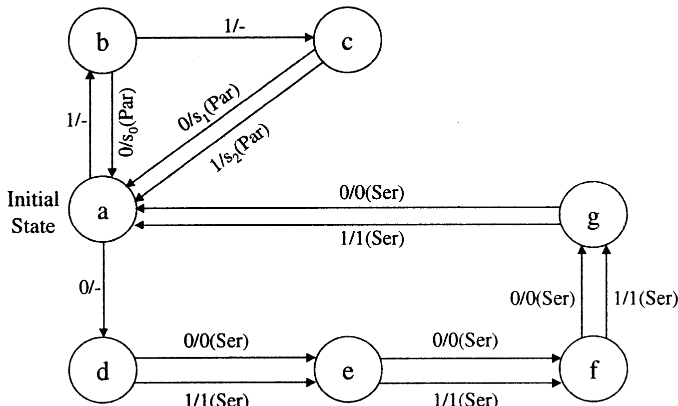

Fig. 9. State transition diagram of the FSM decoder for selected code.

for both full Huffman and selective Huffman encoding using the two algorithms are shown in Section VIII. Also, runtimes are provided to compare the speed of execution of the two algorithms.

VII. Implementation

Once the statistical code has been selected, a decoder for the code is synthesized. One way to implement the decoder is to use a simple FSM. The decoder is driven by the tester clock and one bit of data is received during each tester clock cycle from a tester channel. For a block of size b, the decoder has b data outputs and two main control outputs: $\mathrm{parallel_load}$ (Par) and $\mathrm{serial_load}$ (Ser). These two signals control the loading of data into the serializer when the data has been decoded. The state transition diagram for the decoder FSM can be easily formed from the tree representation of the code. An example is shown in Fig. 9. Note that each codeword has a bit indicating whether the pattern that follows is encoded or not. In this scheme, a 0 is used to indicate "no coding" and a 1 is used to indicate coding. If the first bit of the codeword is a 0, indicating that the next b bits are not coded, then the decoder simply passes the bits through by serially loading them into the serializer. If the first bit of the codeword is a 1, indicating that the subsequent bits form a prefix-free variable length code, then the decoder branches on each bit, one at a time, until it reaches the end of the codeword. At that point it does a parallel load of the appropriate b-bit block into the serializer.

Note that for the sake of clarity, all the control signals between the decoder and the serializer have not been shown in Figs. 3 and 4. Also, the serializer has been shown to be driven by the faster system clock directly, in order to convey the high level idea of the scheme. In practice, the faster system clock driving the serializer is actually controlled by the decoder to synchronize the operations of the decoder and the serializer. Every time the decoder loads data in parallel into the serializer, it enables the system clock for the next b clock cycles to allow the b-bit output to be shifted into the scan chain. Once the b-bits are shifted into the scan chain, the serializer clock is frozen until the decoder is ready to either load in parallel the next decoded output, or ready to shift into the serializer the next bit of an unencoded source symbol. Thus, the serializer is driven in two modes by the decoder. In the faster mode, it is driven by the system clock (controlled by the decoder) to shift b bits into the scan chain. In the slower mode, it is driven by the serializer at the tester clock frequency to shift in one bit of unencoded test data into it.

Another way to implement the decoder is to use a ROM or RAM by placing a further restriction on the selection of the statistical code. The additional restriction is that all codewords must be one of two fixed sizes. If the first bit of the codeword is a 0, then the next b-bits are the block itself. If the first bit of the codeword is a 1, then the next a-bits form an address into a ROM/RAM whose contents contain the corresponding b-bit block. For example, if the block size is 8 ($b = 8$), and the address size is 4 ($a = 4$), then there are $2^8 = 64$ different 8-bit blocks and $2^4 = 16$ of them can be encoded with $(a + 1 = 5)$ bit codewords while the remaining 48 of them would have $(b + 1 = 9)$ bit codewords. The decoding would be done using a 16 $\times$ 8-bit ROM or RAM. If a RAM is used, it could be reused for different cores by simply changing its contents to correspond to the appropriate code for each core. Note also that a RAM that is already present in the functional design could be adapted for this purpose during testing. If the RAM is bigger than what is needed, it can be used by simply padding the high-order address bits with 0's and looking only at the low-order data bits when decoding the statistical codes.

VIII. Experimental Results

The proposed compression/decompression scheme was used to compress test sets for the largest ISCAS'89 [3] circuits assuming full scan. The test sets used for all the experiments in this section are the dynamically compacted test cubes generated by MINTEST [15]. These are the exact same test sets used for the experiments in [6], [7], [14].

The first set of experimental results presented in Table II compares the effectiveness of the various X-filling algorithms discussed in Section VI. Alg1 refers to the exact algorithm, and Alg2 refers to the fast heuristic algorithm. Additionally, results for a third algorithm (referred to as the zero-fill) are also shown. The zero-fill algorithm simply fills the X's with 0's. Note that the zero-fill approach is used by the techniques in [6], [7], [14] as all those techniques eventually compress runs of 0's. Column 1 shows the name of the circuit, column 2 shows the total number of test data bits (not considering the output responses) originally present in the uncompressed test set, and column 3 shows the different block sizes that were used for the compression. Results are shown for two block sizes, 8 and 12. The table entries under the headings "full Huffman coding" show the percentage compression obtained when all the blocks are encoded and those under the heading "selective Huffman coding" show the same when the top n most frequently occurring blocks are encoded. The percentage compression is computed as

$$\frac{(\text{Original Bits-Compressed Bits}) \times 100}{(\text{Original Bits})}.$$

The value of n for the different cases is shown in the first column under "selective Huffman coding". It is explained later on in this section how to choose the right value of n for such cases. It is clearly seen that for most of the cases Alg2 produces results that are almost as good as those produced by Alg1. Note that zero-fill, on the other hand, produces substantially suboptimal results for all the cases and clearly is not the best choice for this scheme.

The experimental results in Table III compare the run times of Alg1 and Alg2 for different block sizes. The zero-fill algorithm is trivial (it simply replaces the X's with O's) and, hence, has not been included in the comparison. The runtimes shown (in CPU seconds) are for a 1-Ghz Pentium III workstation with 1-GB main memory. It is easily seen from

TABLE II
COMPARISON OF DIFFERENT X-FILLING ALGORITHMS FOR COMPRESSION EFFECTIVENESS

Circuit Name	Original Test Data (bits)	Block Size (b)	Full Huffman Coding			Selective Huffman Coding			
			Alg1	Alg2	Zero Fill	# Encoded (n)	Alg1	Alg2	Zero Fill
s5378	23754	8	58.9	57.7	50.7	8	50.1	49.0	43.3
		12	66.0	65.4	56.5	16	55.1	54.1	44.7
s9234	39273	8	59.7	58.0	47.6	7	50.3	48.5	38.5
		12	64.5	63.2	51.3	16	54.2	52.8	38.2
s13207	165200	8	81.0	80.8	77.7	5	69.0	69.0	65.3
		12	85.5	85.3	81.0	14	77.0	76.9	72.6
s15850	76986	8	70.6	69.1	65.7	8	60.0	58.6	55.7
		12	74.8	73.6	68.7	21	66.0	64.7	59.7
s38417	164736	8	64.7	64.3	52.7	7	55.1	54.8	43.7
		12	68.5	68.0	57.2	17	59.0	58.7	45.9
s38584	199104	8	68.0	67.6	61.2	7	58.3	58.1	51.7
		12	72.1	71.3	63.7	18	64.1	63.4	55.8

TABLE III
COMPARISON OF RUN-TIMES (CPU SECONDS) OF ALG1 AND ALG2

Circuit Name	Algorithm	Block Size			
		6	8	10	12
s5378	Alg1	1.1	8.2	40.1	186.0
	Alg2	0.3	1.1	1.9	2.6
s9234	Alg1	1.5	14.4	78.6	382.7
	Alg2	0.3	1.0	1.9	2.6
s13207	Alg1	1.5	12.5	67.7	346.3
	Alg2	0.3	1.0	1.7	2.2
s15850	Alg1	1.7	16.2	91.3	450.2
	Alg2	0.3	1.2	2.3	3.5
s38417	Alg1	1.7	21.1	78.8	715.7
	Alg2	0.4	2.9	3.9	15.8
s38584	Alg1	2.0	33.7	214.8	1122.4
	Alg2	0.5	4.4	11.1	17.8
Average Speed-Up		4.5	9.1	25.1	72.0

TABLE IV
AREA OVERHEAD OF THE DECODER AND SERIALIZER

Circuit Name	Block Size (b)	Full Huffman		Selective Huffman		Serializer
		Num. States ($2^b - 1$)	Area Overhead	Num. States ($n + b$)	Area Overhead	Area Overhead
s5378	8	255	47.9	16	4.0	3.7
	10	1023	61.0	26	10.8	4.7
	12	4095	79.2	28	10.4	5.6
s9234	8	255	30.0	15	4.6	3.0
	10	1023	46.5	26	8.2	3.7
	12	4095	51.1	28	8.6	4.5
s13207	8	255	13.6	13	1.6	1.3
	10	1023	18.9	19	2.8	1.6
	12	4095	23.6	26	3.7	2.0
s15850	8	255	13.0	16	1.3	1.3
	10	1023	19.0	20	3.0	1.7
	12	4095	26.1	33	4.5	2.0
s38417	8	255	6.9	15	0.7	0.4
	10	1023	10.4	27	1.3	0.6
	12	4095	21.3	29	1.3	0.7
s38584	8	255	7.5	15	0.6	0.4
	10	1023	14.4	21	1.0	0.5
	12	4095	19.7	30	1.3	0.7

the results that the runtimes for Alg1 increases rapidly as the block size is increased. However, the increase in runtime for Alg2 is much more controlled. The last row of the table shows the average speed-up that is achieved by Alg2 over Alg1 and is computed as

$$\text{Speed-Up}_{\text{avg}} = \frac{\text{Average Run-Time of Alg 1}}{\text{Average Run-Time of Alg 2}}.$$

For a block size of 12, the average speed-up obtained is as much as 72.

The next set of experimental results shows the hardware overhead for the decoders and the serializer for different block sizes. For the decoders, the hardware overhead is shown for both full Huffman encoding and selective Huffman encoding. In Table IV below, under the column "full Huffman decoder" the number of states of the decoder and the area overhead is shown, and the same for selective Huffman encoding is shown under the column "selective Huffman decoder." The benchmark circuits were synthesized with a single scan chain. The last column in the table shows the area overhead of the serializers. The area overhead for the decoders and the serializers have been shown separately, as the serializer area overhead is the same for both the full Huffman and selective Huffman decoders. Moreover, separating the two area overheads gives a more accurate picture if existing functional hardware on the chip is used to implement the serializer functionality. The area overhead of the decoder (serializer) is computed as

$$\frac{\text{area of decoder (serializer)} \times 100}{\text{area of benchmark circuit}}.$$

As can be seen from Tables II and IV, the code selected for the proposed scheme provides slightly less compression than a Huffman code (the difference becomes less for larger block sizes), but it allows the use of a much simpler decoder. While the number of states for the Huffman decoder grows exponentially, the number of states for the proposed scheme grows linearly. The block size provides an easy way to tradeoff between area overhead and compression with the proposed scheme. Larger block sizes generally give greater compression, but require a more complex decoder and larger serializer.

Next, a procedure for selecting an appropriate value of n (the number of the most frequently occurring blocks that are encoded) is described. Figs. 10 and 11 show the variation of percentage compression with the number of patterns encoded (n) for the selective Huffman code.

An interesting observation that can be made is that initially with an increase in n, the amount of compression increases. The graph reaches a peak at some value for n, and then steadily starts decreasing until all the patterns are encoded. At this point, there is a steep rise in the graph. This can be explained as follows. The patterns are coded in a decreasing order of their frequency of occurrence. After a certain point (the peak), the gains obtained by compressing more patterns are lost due to the increase in the codeword size because of the extra bit added at the begin-

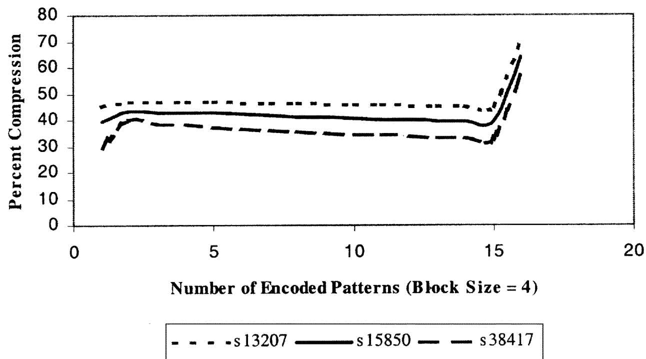

Fig. 10. Variation of compression with n for $b = 4$.

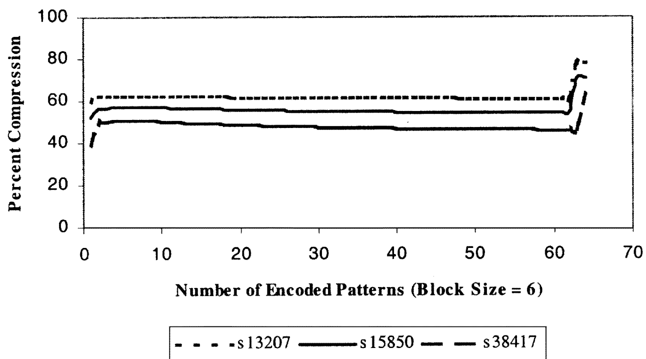

Fig. 11. Variation of compression with n for $b = 6$.

TABLE V
VARIATION OF COMPRESSION PERCENTAGE WITH BLOCK SIZE FOR FULL HUFFMAN ENCODING

Circuit Name	Block Size 4	8	12	16	20	24
s5378	48.2	57.7	65.4	70.7	74.6	78.0
s9234	52.0	58.0	63.2	67.8	71.6	74.6
s13207	69.5	80.8	85.3	87.3	89.1	90.2
s15850	62.2	69.1	73.6	76.6	79.3	81.3
s38417	56.8	64.3	68.0	71.1	74.2	75.0
s38584	59.2	67.6	71.3	74.6	76.8	78.8

ning of the codeword to distinguish coded patterns from those that are not coded. When all the patterns are coded (full Huffman encoding), there is no need for the extra bit at the beginning of each codeword and this greatly increases the amount of compression resulting in the steep slope toward the end of the graph. This trend is observed for all the three different circuits $s13\,207$, $s15\,850$, and $s38\,417$ for the two different block sizes of 4 (in Fig. 10) and 6 (in Fig. 11). Choosing the right value for n is simply a matter of determining the peak from the graphs shown in Figs. 10 and 11. The values of n for the different circuits shown in Table II were all determined based on such graphs for the corresponding circuits.

Table V shows how the percentage compression achieved for full Huffman encoding varies with different block sizes. As mentioned earlier, the amount of compression increases with an increase in the block size and choosing the right value for b is simply a matter of tradeoff between the hardware overhead and the compression desired. The percentage compression shown below for the different block sizes were obtained by using Alg2 for filling the X's. The increase in the amount

TABLE VI
COMPARING TEST DATA COMPRESSION WITH PREVIOUSLY PUBLISHED RESULTS

Circuit Name	Original Test Data (bits)	Golomb [7] % Comp	Golomb [7] *M*	FDR [8] % Comp	FDR [8] *k*	VIHC [14] % Comp	VIHC [14] m_h	Selective Huffman % Comp
s5378	25466	37.1	4	48.0	7	46.9	16	**55.1**
s9234	36309	45.3	4	43.6	6	46.1	16	**54.2**
s13207	167300	79.7	16	**81.3**	9	80.4	16	77.0
s15850	73320	62.8	8	**66.2**	9	64.4	12	66.0
s38417	158080	28.4	4	43.3	10	47.8	16	**59.0**
s38584	191784	57.2	8	60.9	9	59.6	16	**64.1**

of compression is bigger when moving upwards from small block sizes and gradually flattens out at large block sizes. Thus, an optimum value of b would be the point just before the percentage compression curve flattens out.

Three other techniques directly relate to the scheme proposed here. They are compression based on Golomb [6], FDR [7], and variable-input Huffman (VIHC) codes [14]. In the remainder of this section, comparisons between these different techniques in terms of compression efficiency, hardware overhead of the decompressors, and test application time reduction will be discussed.

Table VI shows the amount of compression obtained using the four different techniques. The exact same test set (MINTEST dynamically compacted test cubes) was used for all the methods. Columns 1 and 2 in Table VI report the circuit name and the number of bits in the original test data set. The numbers for selective Huffman coding in Column 3 are taken from Table II for Alg1 and block size $b = 12$ as a sample representative. Although more compression can be obtained with larger block sizes, the area overhead increases. The block size of 12 will be used as a representative for selective Huffman for all the comparisons in this section. The number of blocks actually encoded for this scheme can be obtained from Table II. The best compression obtained using Golomb coding [6] is shown in the next two columns. The value of the parameter m (group size) for which this compression is obtained is also shown in the table next to the percentage compression. Similar results are also shown in the table for the FDR [7] and the VIHC codes [14]. For the FDR codes, the value of the parameter k, is determined by the longest run of 0's and is shown in the table next to the percent compression. For the VIHC codes, the results shown are the best compression percentage obtained using different values for the parameter m_h (the maximum run of 0's allowed in any block). The values of the parameter m_h tried were 4, 6, 8, 10, 12, 14, and 16. For all the other techniques (Golomb, FDR, and VIHC), the zero-fill algorithm was used for filling the X's as that maximizes the amount of compression using those methods. This ensures that the comparison is fair. For each of the circuits, the method that produces the best compression is highlighted in bold font. It can be seen that the selective Huffman code produces the best compression for all the circuits except s13207 and s15850, for which the best compression is produced by the FDR code. However, for s15850 the improvement of the FDR code over selective Huffman is only 0.2%.

The area overhead of the decoders for the different compression methods are compared next in Table VII. The area overhead is computed as explained earlier for the results shown in Table IV. The partic-

TABLE VII
COMPARING DECODER AREA OVERHEAD

Circuit Name	Decoder Area Overhead (%)			
	Golomb [7]	FDR [8]	VIHC [14]	Selective Huffman
s5378	**4.0**	7.5	5.8	16.0
s9234	**3.2**	5.5	4.6	13.0
s13207	4.1	3.1	**2.2**	5.7
s15850	**2.0**	3.2	2.3	6.5
s38417	**0.5**	1.1	0.7	2.0
s38584	**0.7**	1.1	**0.7**	2.0

TABLE VIII
COMPARING LOWER BOUNDS ON $f_{\rm sys}/f_t$ TO OBTAIN MAXIMUM TAT REDUCTION

Circuit Name	f_{sys}/f_T to Obtain Max. TAT Reduction			
	Golomb [7]	FDR [8]	VIHC [14]	Selective Huffman
s5378	**4**	128	8	6
s9234	**4**	64	5.3	6
s13207	16	512	16	**6**
s15850	8	512	12	**6**
s38417	**4**	1024	16	6
s38584	8	512	8	**6**

ular decoder configuration for each of the individual different methods is determined by the results of Table VI above. For example, since in Table VI, a block size of 12 has been chosen as a representative for selective Huffman coding, all decoder area overheads for selective encoding includes area overhead for a 12-bit serializer. Similarly for the other codes, the decoder configuration is determined by the respective parameters shown in Table VI for each of the circuits (parameter m for Golomb codes, k for FDR codes, and m_h for VIHC codes). The area overhead computed for each of the following methods includes all the hardware structures involved in the decoding process (the core FSM decoders along with all the counters, serializers etc.). Note that the actual area overhead for each of the techniques could actually be less in certain cases if existing functional circuitry on-chip is reused to implement some of the decoder structures like counters, serializers, etc. Also, note that the area overhead could be reduced by using different parameters for each code, but that would come at the cost of less compression. The technique that produces the least decoder area overhead is highlighted in bold font for each of the test cases.

Finally, the total test application time (TAT) reduction that can be achieved using the different compression techniques will be discussed. The frequencies of the system clock and the tester clock greatly affect the TAT. The question addressed here is "what is the maximum reduction in TAT possible for each of these methods, and what are the necessary conditions for achieving the maximum reduction?" For the discussion that follows, it will be assumed that the tester is testing a single core at any given time and, without loss of generality, it will be assumed that the core has a single scan chain which is driven by the decoder, which, in turn, is driven by a single channel from the tester.

In a scheme where compression/decompression is used to reduce the test data volume stored on the tester, the total time required to transfer the reduced test data from the tester to the chip decreases. If the tester never stops shifting in data during the entire duration of the test data transfer, then the maximum possible TAT reduction that can be obtained by any of these techniques is the same percentage as the amount of test data reduction achieved by the given technique, however, there are conditions under which this can be achieved. There are three key parameters. The size of the smallest codeword $L_{\min}$, the size of the largest decoded output $L_{\max}$, and the ratio of the system clock frequency $(f_{\rm sys})$ to the tester clock frequency (f_T). $f_{\rm sys}/f_T$ defines how fast the system clock is relative to the tester clock. Generally, the greater the value of $f_{\rm sys}/f_T$, the faster the data can be decoded and applied to the scan chain. In the next few paragraphs, the hardware decompression mechanism for each of the four techniques will be discussed in greater detail, and factors that affect how fast the data can be decoded in each case will be examined. A lower bound on $f_{\rm sys}/f_T$ will be derived for each of the 4 techniques that will allow maximum reduction in TAT.

The analysis for selective Huffman coding has already been discussed earlier in Section V. From (i) in Section V, a lower bound on $f_{\rm sys}/f_T$ can be obtained that will allow the maximum reduction in TAT (by ensuring that the tester never has to wait for the decoder to finish decoding the previous codeword). This lower bound is given by $f_{\rm sys}/f_T \geq (b/L_{\min})$ where b is the block size.

For Golomb coding, the parameter m (the group size) determines the lower bound on $f_{\rm sys}/f_T$. When decoding the prefix, every time the decoder sees a 1, it has to shift into the scan chain of the core a sequence of m 0's. The Golomb method uses a $\lceil \log_2 m \rceil$-bit counter for this purpose, which can be operated by the faster system clock. Thus, $f_{\rm sys}/f_T \geq m$ ensures that the tester never has to wait for the decoder to finish decoding the previous codeword.

For FDR codes with a parameter k (which is determined by the longest run of 0's in the original test set), the worst case is when the decoder has to shift into the scan chain 2^k decoded output bits after it has just finished shifting the last bit of the tail into the k-bit counter. Thus, the counter has to finish counting down 2^k times before the decoder receives the first prefix bit of the next codeword. Hence, the lower bound is given by $f_{\rm sys}/f_T \geq 2^k$.

For VIHC codes with parameter m_h, the analysis is very similar to the selective Huffman codes. In this case, after the decoder has decoded the codeword, it loads the binary code corresponding to the Huffman code into a $\lceil \log_2 m_h \rceil$-bit counter. The counter starts counting down at this point and, in the worst case, has to finish counting m_h times before the decoder is ready with the next decoded output. Again, in the worst case, the decoder will be ready with the next codeword after $L_{\min}$ tester cycles. Hence, the lower bound is given by $(f_{\rm sys}/f_T \geq mh/L_{\min})$.

The value of the lower bound on $f_{\rm sys}/f_T$ is shown in Table VIII. The value of the parameters for each method is the same as shown in Table VI earlier. If $f_{\rm sys}/f_T$ is greater than or equal to the lower bound shown in Table VIII for each method, then the reduction in TAT will be equal to the test data compression for that method given in Table VI.

Note that if $f_{\rm sys}/f_T$ is greater than the lower bound, it does not improve the reduction in TAT. In such cases, the decoder has to occasionally wait for the tester to supply it with encoded data. The higher the lower bound on $f_{\rm sys}/f_T$, the more difficult it is to achieve the maximum reduction in TAT (i.e., where the TAT reduction is equal to the test data compression).

IX. CONCLUSION

Statistical coding provides a powerful way to compress test data. It provides a twofold advantage in both reducing the amount of test data that needs to be stored on the tester and reducing the time for transferring test data from the tester to the CUT. The scheme described in this paper addresses this problem by selecting a "simple-to-decode" statistical code for a particular test set. Results indicate that a small decoder (compared to a full Huffman decoder) can be used to provide compression near that of an optimal Huffman code.

ACKNOWLEDGMENT

The authors would like to thank Prof. K. Chakrabarty and his students for providing the MINTEST test cubes that were used for the experimental results in this paper.

REFERENCES

[1] J. Aerts and E. J. Marinissen, "Scan chain design for test time reduction in core-based ICs," in *Proc. Int. Test Conf.*, 1998, pp. 448–457.
[2] I. Bayraktaroglu and A. Orailoglu, "Test volume and application time reduction through scan chain concealment," in *Proc. Design Automation Conf.*, 2001, pp. 151–155.
[3] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *Proc. Int. Symp. Circuits Syst.*, 1989, pp. 1929–1934.
[4] M. L. Bushnell and V. D. Agrawal, *Essentials of Electronic Testing For Digital, Memory, And Mixed-Signal VLSI Circuits*. Norwell, MA: Kluwer, 2000.
[5] A. Chandra and K. Chakrabarty, "Test data compression for system-on-a-chip using Golomb codes," in *Proc. VLSI Test Symp.*, 2000, pp. 113–120.
[6] ——, "Efficient test data compression and decompression for system-on-a-chip using internal scan chains and Golomb coding," in *Proc. Design, Automation, Test Eur.*, 2001.
[7] ——, "Frequency-directed run-length codes with application to system-on-a-chip test data compression," in *Proc. VLSI Test Symp.*, 2001, pp. 42–47.
[8] ——, "Reduction of SOC test data volume, scan power and testing time using alternating run-length codes," in *Proc. Design Automation Conf.*, 2002, pp. 673–678.
[9] R. Chandramouli and S. Pateras, "Testing systems on a chip," *IEEE Spectrum*, pp. 42–47, Nov. 1996.
[10] D. Das and N. A. Touba, "Reducing test data volume using external/LBIST hybrid test patterns," in *Proc. Int. Test Conf.*, 2000, pp. 115–122.
[11] R. Dorsch and H.-J. Wunderlich, "Reusing scan chains for test pattern decompression," in *Proc. European Test Workshop*, 2001, pp. 124–132.
[12] ——, "Tailoring ATPG for embedded testing," in *Proc. Int. Test Conf.*, 2001, pp. 530–537.
[13] A. El-Maleh, S. Al-Zahir, and E. Khan, "A geometric-primitives-based compression scheme for testing systems-on-a-chip," in *Proc. VLSI Test Symp.*, 2001, pp. 54–59.
[14] P. Gonciari, B. M. Al-Hashimi, and N. Nicolici, "Improving compression ratio, area overhead, and test application time for systems-on-a-chip test data compression/decompression," in *Proc. Design Automation Test Eur.*, 2002, pp. 604–611.
[15] I. Hamzaoglu and J. H. Patel, "Test set compaction algorithms for combinational circuits," in *Proc. Int. Conf. Computer-Aided Design*, 1998, pp. 283–289.
[16] ——, "Reducing test application time for full scan embedded cores," in *Proc. Int. Symp. Fault-Tolerant Comput.*, 1999, pp. 260–267.
[17] D. Heidel, S. Dhong, P. Hofstee, M. Immediato, K. Nowka, J. Silberman, and K. Stawiasz, "High speed serializing/de-serializing design-for-test method for evaluating a 1 GHz microprocessor," in *Proc. VLSI Test Symp.*, 1998, pp. 234–238.
[18] D. A. Huffman, "A Method for the construction of minimum redundancy codes," in *Proc. IRE*, vol. 40, 1952, pp. 1098–1101.
[19] H. Ichihara, K. Kinoshita, I. Pomeranz, and S. M. Reddy, "Test transformation to improve compaction by statistical encoding," in *Proc. Int. Conf. VLSI Design*, 2000, pp. 294–299.
[20] H. Ichihara, A. Ogawa, T. Inoue, and A. Tamura, "Dynamic test compression using statistical encoding," in *Proc. Asian Test Symp.*, 2001, pp. 143–148.
[21] M. Ishida, D. S. Ha, and T. Yamaguchi, "COMPACT: a hybrid method for compressing test data," in *Proc. VLSI Test Symp.*, 1998, pp. 62–69.
[22] V. Iyengar, K. Chakrabarty, and B. T. Murray, "Built-in self testing of sequential circuits using precomputed test sets," in *Proc. VLSI Test Symp.*, 1998, pp. 418–423.
[23] A. Jas and N. A. 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.
[24] A. Jas, J. Ghosh-Dastidar, and N. A. Touba, "ScanVector compression/decompression using statistical coding," in *Proc. VLSI Test Symp.*, 1999, pp. 114–120.
[25] A. Jas and N. A. Touba, "Using an embedded processor for efficient deterministic testing of systems-on-a-chip," in *Proc. Int. Conf. Computer Design*, 1999, pp. 418–423.
[26] A. Jas, B. Pouya, and N. A. Touba, "Virtual scan chains: a means for reducing scan length in cores," in *Proc. VLSI Test Symp.*, 2000, pp. 73–78.
[27] A. Jas, C. V. Krishna, and N. A. Touba, "Hybrid BIST based on weighted pseudo-random testing: a new test resource partitioning scheme," in *Proc. VLSI Test Symp.*, 2001, pp. 114–120.
[28] A. Khoche, E. Volkerink, J. Rivoir, and S. Mitra, "Test vector compression using EDA-ATE synergies," in *Proc. VLSI Test Symp.*, 2002, pp. 97–102.
[29] C. V. Krishna, A. Jas, and N. A. Touba, "Test vector encoding using partial LFSR reseeding," in *Proc. Int. Test Conf.*, 2001, pp. 885–893.
[30] J. Rajski and J. Tyszer, "Modular logic built-in self-test for IP cores," in *Proc. Int. Test Conf.*, 1998, pp. 313–321.
[31] P. Rosinger, P. T. Gonciari, B. M. Al-Hashimi, and N. Nicolici, "Simultaneous reduction in volume of test data and power dissipation for systems-on-a-chip," *Electron. Lett.*, vol. 37, pp. 1434–1436, Nov. 22, 2001.
[32] M. Sugihara, H. Date, and H. Yassura, "A novel test methodology for core-based system LSI's and a testing time minimization problem," in *Proc. Int. Test Conf.*, 1998, pp. 465–472.
[33] T. Yamaguchi, M. Tilgner, M. Ishida, and D. S. Ha, "An efficient method for compressing test data," in *Proc. Int. Test Conf.*, 1997, pp. 191–199.
[34] Y. Zorian, "Test requirements for embedded core-based systems and IEEE P1500," in *Proc. Int. Test Conf.*, 1997, pp. 191–199.