On-Chip Test Generation Using Linear Subspaces

Ramashis Das, Igor L. Markov, John P. Hayes
{ramashis, imarkov, jhayes}@eecs.umich.edu
Advanced Computer Architecture Laboratory, University of Michigan,
Ann Arbor, MI 48109, USA

Abstract

A central problem in built-in self test (BIST) is how to efficiently generate a small set of test vectors that detect all targeted faults. We propose a novel solution that uses linear algebraic concepts to partition the vector space of tests into subspaces (clusters). A subspace is defined by a compact set of basis vectors. We give an algorithm to compute sets of basis vectors defining the clusters. We also describe a low-cost logic circuit based on Gray codes that reproduces the subspaces from these basis vectors. Experimental results are presented which show that this approach reduces on-chip hardware overhead and test application time, while also guaranteeing full fault coverage.

1 Introduction

A typical built-in self test (BIST) circuit consists of a controller, a test pattern generator (TPG) that feeds test vectors to the circuit under test (CUT), and a response analyzer that verifies the test responses. The efficiency of the TPG in generating good test vectors largely determines the performance of the BIST. Hardware overhead, testing time and the fault coverage are the performance metrics of the TPG. The hardware structure of a TPG can be broadly divided into a state controller, a combinational mapping circuit and an optional memory element or ROM (Fig. 1). The state controller *SC* holds the current state of TPG in a register *SR*, and an additional circuit C_{state} determines the next state. The mapping logic *ML* decodes the state into test inputs for the CUT. The ROM *M* is used to store some predetermined test data. As we discuss, all TPG methods try to reduce or remove one or more of these blocks. For example, storing a minimal set of complete tests can get rid of all blocks except the ROM *M*. It can also guarantee full fault coverage, but may require very large hardware overhead in terms of ROM size.

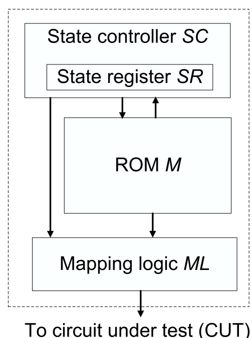

Figure 1: General structure of a test pattern generator.

The most common example of a test pattern generator is the linear feedback shift register (LFSR) which has no explicit *M* or *ML* [15]. The next state is derived from an XOR-network inside the LFSR which comprises the C_{state} part. LFSRs are popular due to their small size and ability to generate large spaces of pseudo-random tests. A major problem with LFSRs is that they need to generate many test vectors to achieve high fault coverage. Many methods in the literature propose to speed up the LFSR with additional hardware. A recent paper by Chatterjee and Pradhan [2] introduces the concept of Generalized LFSRs (GLFSRs) that generate test vectors over the Galois field $GF(2^d)$. GLFSRs achieve nearly complete fault coverage by increasing the dimension of the LFSR-state by one or more. A further improvement on GLFSRs is discussed in [5]. Though these low-hardware overhead modifications to LFSRs improve fault coverage with a fairly small number of test patterns, they do not guarantee complete fault coverage.

It has been observed that the presence of "pseudo-random pattern resistant" faults in the CUT makes it difficult to achieve full fault coverage. A primitive polynomial-based LFSR running through all its $2^n - 1$ states can guarantee complete fault coverage, but takes time that is exponential to the test size *n*. (A small modification can add the missing all-0 state, if required.) Reseeding of LFSRs [12] is one way to achieve complete fault coverage that avoids visiting all states. In this technique, the LFSR is run for a predetermined number of cycles and then restarted with a different state (a seed). This process is repeated several times. The seeds can be stored in a small ROM *M* accessible to the sequential machine that controls the LFSR. A similar approach is to use a multiple-polynomial LFSR [11] where one changes the polynomial function of the LFSR after a predetermined number of cycles. Both approaches use *M* to store some critical state information. Weighted random pattern generators [7] provide yet another LFSR-based method that uses extra hardware to achieve full fault coverage.

While LFSRs form a class of linear logic circuits, linear algebraic methods as such are rarely exploited explicitly in test pattern generation. The earliest example appears to be [1] which proposes using a *k*-bit counter feeding an XOR array *C* to generate a given test set *T*, recognizing that *C* performs a linear transformation (Fig. 2). Although not involving linear algebra directly, [8] is similar in spirit, in that the authors treat a test set as a matrix, and use column operations like permutation and complementation to convert it to a form that is readily generated by an ordinary binary counter. Similarly, the literature on pseudo-exhaustive testing distinguishes partitions of test vectors based on independence among primary input variables of the CUT. Table 1 classifies the various proposed TPG methods.

Many testing techniques, espescially those based on external ATE, implement compression methods. Some of these

TPG method	ROM contents M	State controller SC	Mapping logic ML	Reference(s)
Pre-stored tests	Complete test set	-	-	-
Simple LFSR-based	-	LFSR	-	[15]
GLFSR-based	-	Generalized LFSR	-	[2], [5]
Weighted random	Weights	LFSR	Custom	[7]
Reseeding/multiple polynomial LFSR	Seeds/polynomials	LFSR	Custom	[10]
Bit flipping	-	LFSR	Bit flipping logic	[13]
Linear transformations	-	Binary counter	XOR array	[1]

Table 1: Classification of hardware-implemented TPG methods.

techniques ([6], [17], [18]) use clustering to minimize storage requirements and the amount of external data required for testing, hence reducing the testing time.

In this work we develop a TPG methodology based on dividing the test vector space into clusters (subspaces) of small rank. We propose an efficient pseudo-exhaustive enumeration of subspaces, based on Gray-code counters which leads to a very efficient TPG design. We explore possible uses of such hardware in BIST to generate a desired test set.

The remainder of this paper is organized as follows. Section 2 introduces the algebraic operations used to define linear spaces of test vectors. Sections 3 and 4 describes our TPG design to enumerate all vectors in a linear subspace defined by sets of basis vectors. An algorithm for clustering of test vectors into linear subspaces is proposed in Section 5. Experimental results demonstrating the performance of the proposed TPG and comparison with existing TPGs are presented in Section 6. Conclusions and future work are discussed in Section 7.

2 Theoretical Framework

Consider an n-dimensional vector space V over the field $F_2 = \{0,1\}$, in which the field operations $+$ and $\times$ are bit-wise logic XOR and AND operations, respectively. Test vectors with n bits can therefore be thought of as elements of V. A basis for V is a smallest set of linearly-independent n-bit vectors that can generate the entire space by bit-wise XOR operations. For $n = 3$, $V_3 = \{000,001,010,011,100,101,110,111\}$, and an example basis is $B_3 = \{001,010,100\}$. A subset of B_3 spans a subset V' of V that corresponds to a subspace or *cluster* of V. For example, $V'_3 = \{000,001,010,011\}$ is a cluster of V_3 generated by $B'_3 = \{001,010\}$. Thus, the cluster is captured by a smaller set of basis vectors. We exploit this idea to design TPGs that use small basis sets to generate clusters with full fault coverage.

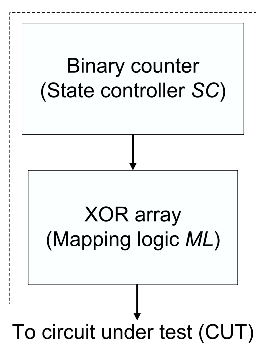

Figure 2: Test set embedding using an XOR array [1]

Let us now look at an example that further explains the algebra underlying our approach. Consider the c499 single-error-correcting circuit from the ISCAS-85 benchmark suite. Applying the ATALANTA ATPG program [14] to the gate-level version of this circuit produces a set S of 53 41-bit test vectors that detect all non-redundant faults. These vectors, when arranged in the order of number of faults covered, form the following array:

1:	**01100111100110000011000101011010010110000**
2:	**10110001001010111111110110110001111101110**
3:	**11010101111010000001010110111000111010101**
4:	**10101110000100001101010110001011101001101**
5:	**00100010010010110011111101011001011101011**
6:	**00100000011110100000000111110111001011111**
7:	**11111101100110110100000001010000000011010000**
8:	**11111011100101111010010000111100001010001**
9:	00001110110000111111001110101001110011100
10:	11010101100010110001010111111011000011100
11:	10011010111000011111110101100011111001110
⋮	⋮
52:	10010100100011010111010001110111001011100
53:	11111111000000000110100100100001101101101

Simulation experiments show that the linear space V'_{41} spanned by the highlighted vectors is enough to achieve full fault coverage. These eight linearly independent vectors form a basis B'_{41}; hence a TPG circuit that can generate the entire space spanned by these basis vectors will ensure complete fault coverage. This set of basis vectors is 6.6 times smaller than the original test set S. Figure 3*a* shows a TPG design that stores the complete test set S in the ROM M, and Fig. 3*b* shows the proposed TPG where only B'_{41} is stored in the ROM M', which is considerably smaller than M.

The cluster V'_{41} spanned by the 8 basis vectors B'_{41} contains $2^8 = 256$ vectors, not all of which are important in detecting faults. Often we can generate several clusters of smaller size which result in smaller test sets. For c499, full coverage can be achieved by combining four separate bases of rank 5. Though this requires storing 12 extra vectors, the test length is only $128 = 4 \times 2^5$. Note the similarity of producing multiple clusters to the reseeding of an LFSR, where a seed represents a consecutive subset of vectors (cluster) generated by the LFSR.

A ROM that stores k n-bit basis vectors (Fig. 3*b*) v_0, v_1, …, v_{k-1} implicitly represents the linear subspace V'_n spanned by these vectors. Properly selected, this subspace contains many additional test patterns which are linear combinations of the k stored vectors. Consider a k-bit integer m as the bit-string $m_{k-1}m_{k-2}\ldots m_1 m_0$. A linear combination of the stored vectors is given by $\sum_{j=0}^{k} m_j v_j$, where the binary coefficient m_j implies the inclusion (if $m_j = 1$) or exclusion (if $m_j = 0$) of vector v_j in the linear combination. By iterating over all 2^k values of m, we can generate every vector in V'_n. A simple way to generate a cluster from a basis

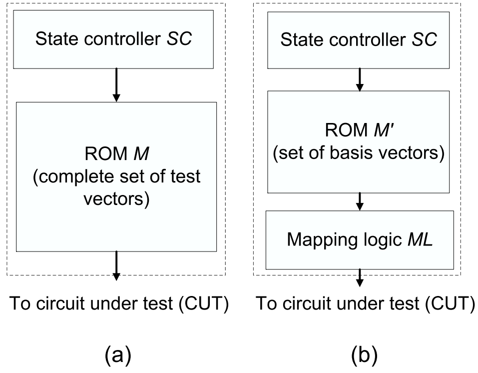

Figure 3: (a) TPG design storing a pre-computed set of test vectors. (b) Proposed TPG design (ROM M' can be replaced by combinational logic)

of rank k is to use a k-bit binary counter which acts as the state controller and an explicit summation implemented by k controlled bit-wise XOR operations (the mapping logic).

A similar concept of algebraic basis is used by Akers and Jansz [1]. Instead of storing the basis vectors, they construct an array of XOR gates in which each row represents a basis vector: the presence of XOR gate implies 1, and the absence implies 0. Each row acts as input to the next row of XORs and the second input of the XOR is the binary counter bit corresponding to the row. Note that this cluster-generating hardware outputs the basis vectors when the binary counter value has a "one-hot" format $(000\ldots01, 000\ldots10, \ldots, 100\ldots00)$. Thus, their TPG hardware (Fig. 2) generates an explicit modulo-2 sum (same as n-bit XOR) of the basis vectors,

$$X_b = \sum_{j=0}^{k} b_j v_j,\ 0 \leqslant b \leqslant 2^{k-1} \tag{1}$$

where b represents the integer value of the counter. In other words, the current state of the TPG. b_j is the jth bit of the binary counter and v_j is the jth basis vector. The modulo-2 sum X_b represents a test vector which is fed to the circuit under test. We can clearly see that the binary counter acts as the state register (Fig. 1) that runs the TPG through the 2^k states, and the XOR array is the mapping logic ML.

3 Proposed TPG Design

As discussed in Section 2, Eq. (1) gives a fairly straightforward implementation of a space-generating method based on a binary counter (Fig. 3b). This requires several layers of XOR gates, resulting in large chip area and delay. In our proposed implementation, we avoid this overhead by using a Gray-code counter instead of a binary counter, as explained below.

We can re-write Eq. (1) as follows:

$$X_g = \sum_{j=0}^{k} g_j v_j,\ 0 \leqslant g \leqslant 2^{k-1} \tag{2}$$

where g represents the output of a Gray-code counter. By definition, two consecutive Gray-code counts g and $g+1$ differ by only one bit. Hence there is just one bit position h such that $g_h \neq (g+1)_h$, and for all $q \neq h, g_q = (g+1)_q$. This implies that X_g and X_{g+1} differ only in the inclusion or exclusion of some basis vector v_h, that is,

$$X_{g+1} = X_g \oplus v_h \tag{3}$$

where h represents the bit position in which two consecutive Gray-code counts differ, i.e. a "one-hot" Gray-code count. This expresses the relation between test vectors in terms of the current test vector and a basis vector that is selected using the one-hot Gray code. Table 2 shows the application of Eq. (3) to a basis $B'_4 = \{0001, 0010, 0100\}$ of rank $k = 3$. Note that the last column contains all the vectors in the subspace defined by B'_4.

Gray-code count (g)	One-hot Gray-code count	Flipping bit (h)	X_g	v_h	X_{g+1}
000	-	-	0000	-	0000
001	001	0	0000	0001	0001
011	010	1	0001	0011	0010
010	001	0	0010	0001	0011
110	100	2	0011	0101	0110
111	001	0	0110	0001	0111
101	010	1	0111	0011	0100
100	001	0	0100	0001	0101

Table 2: Evaluation of Eq. (3) on basis $B'_4 = \{v_0 = 0001, v_1 = 0011, v_2 = 0101\}$

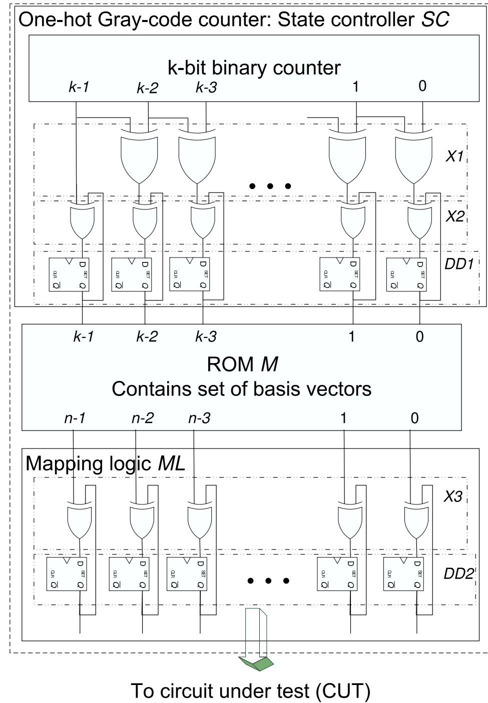

Figure 4: Hardware to generate a cluster (subspace) from a basis stored in the ROM.

Our proposed TPG design implements Eq. (3) rather than the much more complex Eq. (1) using a one-hot Gray-code counter to obtain h. The TPG includes a test-vector register, initialized to $000\ldots0$, whose bits can be simultaneously XORed with ROM output v_h (ML in Fig. 4). The ROM address h represents the position of the 1 in the k-bit one-hot Gray count. Such sequences can be generated by

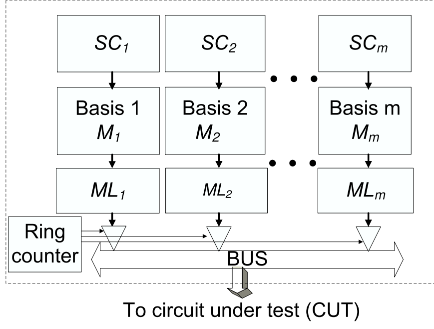

Figure 5: TPG hardware to generate multiple clusters.

a k-bit binary counter, whose output is converted to Gray code using $k-1$ two-input XOR gates and registered. The bit-wise XOR of two consecutive Gray codewords gives the required ROM address in the one-hot form. This part of the circuit controls the state of the TPG (*SC* in Fig. 4). The one-hot address generated by *SC* makes the traditional ROM-address decoder unnecessary and allows the ROM to accommodate arbitrary k values, not just powers of two. Figure 4 shows more details of the TPG hardware. It is clear that we need 3 layers of logic having $k-1$, k and n of XOR gates ($X1$, $X2$ and $X3$ respectively), and 2 layers of logic having k and n D flip-flops ($DD1$ and $DD2$ respectively).

As mentioned, it is sometimes more efficient to break the entire space into smaller subspaces, which requires generation of subspaces from different basis sets. This can be implemented by connecting the outputs of each space generator to a bus (Fig. 5). Individual one-hot Gray-code counters can be enabled/disabled using a shared ring counter. A more efficient design reuses the test-vector register and the XOR hardware. In particular, all one-hot Gray-code counters can then be consolidated in a single counter. A TPG for m subspaces requires just an additional $\log m$ bits in the binary counter used to generate the Gray code.

4 Improved TPG Design

We now look into ways of improving the design of Fig. 4. The layers $X3$ and $DD2$ XOR the current test vector with the next basis vector. A 1 in the basis vector will flip the test vector whereas a 0 will cause no change. Layers $X3$ and $DD2$ can be reduced to a layer of n T (toggle) flip-flops with the ROM output feeding the toggle input of the flip-flops. This reduces the cost of the *ML* part of our TPG as T flip-flops have area cost similar to D flip-flops.

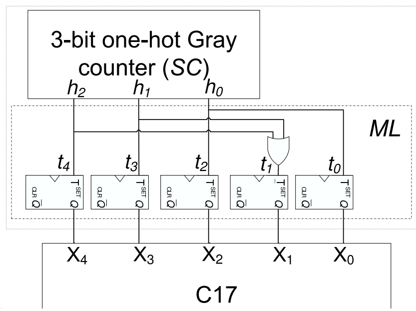

Figure 6: Removing the ROM from the TPG for c17.

Consider the ISCAS-85 c17 benchmark circuit. A basis for complete fault coverage is:

$$I = \begin{matrix} v_2 \\ v_1 \\ v_0 \end{matrix} \begin{bmatrix} 10010 \\ 01111 \\ 10101 \end{bmatrix} \quad (4)$$

To handle the 3 basis vectors, we need a small ROM and a 3-bit binary counter. For clarity, we name the TPG signals as shown in Table 3. We can do elementary row operations (as in diagonalization) to reduce the number of 1s in I to get the matrix:

$$T = \begin{matrix} v_2 \\ v_1 \\ v_0 \end{matrix} \begin{bmatrix} 10010 \\ 01010 \\ 00101 \end{bmatrix} \quad (5)$$

The basis T is stored in the ROM; a vector is selected when the corresponding bit in the one-hot Gray-count is 1. Thus, basis vector v_2 is selected only when h_2 is 1. Bit X_4 is flipped only when v_2 is selected since no other vector in T has a 1 in its MSB position (see Eq. (5)). Thus, we can directly connect h_2 to t_4. Continuing with the same reasoning, h_1 connects directly to t_3, h_0 to t_2 and h_0 to t_0 (Fig. 6).

TPG component	Output signal names				
3-bit binary counter			b_2	b_1	b_0
3-bit Gray-code counter			g_2	g_1	g_0
3-bit one-hot Gray-code counter			h_2	h_1	h_0
5-bit T flip-flop register	t_4	t_3	t_2	t_1	t_0
5-bit test vector	X_4	X_3	X_2	X_1	X_0

Table 3: Notation for various TPG signals in the c17 example.

The handling of t_1 is somewhat complex. X_1 is flipped whenever one of v_1 or v_2 is selected (Eq. (5)), i.e., $t_1 = 1$ whenever $h_1 \vee h_2 = 1$. Hence we can make $t_1 = h_1 \vee h_2$. Thus, we can eliminate the entire ROM by connecting the one-hot encoded Gray counter to the T flip-flops via OR gates. Now, we know that h_i is 1 when g_i flips, and also that X_j flips when t_j is 1. Thus, for an h_i that connects directly to t_j, we can connect g_i to X_j, getting rid of the hardware to generate the one-hot encoded Gray count and the T flip-flops. For cases where t_k is a function of several h_ls, we can construct a combinational circuit to implement the corresponding X_k. In this example, it is found that $X_1 = b_1$. Figure 7 shows the final TPG hardware for c17.

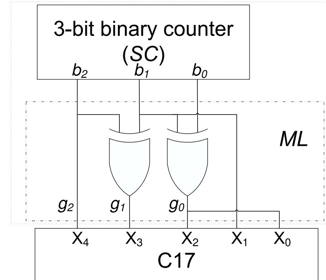

Figure 7: The final TPG for c17.

Let us now discuss the TPG design for a general n-input CUT. Suppose a basis B'_n of k vectors has been computed such that the corresponding subspace achieves complete fault coverage. We can use the design of Fig. 4 to generate the cluster defined by B'_n. For the first design improvement, we replace the $X3$ and $DD2$ layers by T flip-flops, and the ROM by a layer of OR logic. We then apply

Benchmark circuits	No. of inputs	ATALANTA No. of test patterns (n_1)	Proposed Method Max. cluster size	No. of basis vectors (n_2)	Total no. of clusters	Compression ratio (n_1/n_2)	Test size
c432	36	52	9	9	1	5.78	256
c499	41	53	8	8	1	6.63	256
c880	60	58	9	13	2	4.46	527
c1355	41	86	10	10	1	8.60	1024
c1908	33	115	11	14	2	8.21	2055
c2670	233	101	11	40	4	2.53	6269
c3540	50	144	12	14	2	10.29	4099
c5315	178	116	11	11	1	10.55	2048
c6288	32	31	7	7	1	4.43	128
c7552	207	212	13	41	4	5.17	24577

Table 4: Experimental results showing the hardware requirements, compression ratio and test length for full fault coverage.

elementary row operations to the $k \times n$ basis matrix to reduce the number of 1s. The input to each T flip-flop is formed by the OR of a subset of one-hot Gray count bits. For example, $t_i = h_j \vee h_l \vee \ldots$ if the ith column in the matrix has 1s in rows $j, l \ldots$ (similar to Fig. 6). Further hardware modification reduces the entire mapping logic *ML* to a layer of XOR gates as in Fig. 7. Any of these cluster-generating implementations can be scaled up to generate multiple clusters from multiple sets of basis vectors as discussed in Sec. 3.

Table 4 lists the performance of our approach in terms of the hardware requirements and test times for various ISCAS-85 circuits. We define *compression ratio* (column 7) as the ratio of the size of a complete set of test vectors generated by ATALANTA (column 3) to the number of basis vectors required to generate subspace(s) that guarantee complete fault coverage. In other words, it is a measure of the reduction in storage requirement in the ROMs M and M' shown in Fig. 3. We can see a high compression ratio for most of the benchmark circuits, the best result being that for c5315 with a compression ratio of over 10. Similar results are obtained for the ISCAS-89 circuits but are not given here due to lack of adequate comparision data in the prior literature.

5 Subspace Selection

To employ the TPG design proposed in Sec. 3, we need a way to compute one or more (preferably small) basis sets that define a collection of subspaces which guarantees complete fault coverage. One way of getting a basis is by generating random basis vectors and using a fault simulator to compute the fault coverage. We can then use improvement in fault coverage as a criterion to select or reject a vector. Circuit partitioning can be performed to get sub-circuits with small n, and generate a basis for each of these separately. Another way is to start with a complete set of test vectors and obtain a basis from it.

We implement a heuristic algorithm that selects a vector from a set of test vectors S such that its addition to the existing basis leads to maximum improvement in fault coverage. Note that adding a vector to an existing basis of rank k adds 2^k new vectors to the subspace. We start by running ATALANTA with the list of faults not detected by the all-zero vector $000\ldots0$ (which belongs to every subspace) to get the first set S_1. At each subsequent step, we run ATALANTA with a list of undetected faults to get a set S_i. We choose a vector v from S_i that maximizes the coverage of the new basis. This is continued until all non-redundant faults are detected. We will see in Sec. 6 that this fast heuristic method for basis generation produces good results.

Another important aspect of subspace selection is the maximum cluster size. A cluster of k basis vectors requires 2^k clock cycles to generate the corresponding subspace. To limit cluster size, we experimented with various values of k $(7, 8 \ldots 14)$ and observed that an increase in subspace size decreases test time at the cost of ROM size. Thus an optimal value of cluster size needs to be chosen for a particular target circuit. A simple improvement to this algorithm is obtained by decreasing the maximum cluster size as the number of clusters increase. As the first cluster of test vectors covers most of the faults, the size of subsequent clusters can be reduced. This, in turn, reduces the testing time by reducing the number of patterns generated.

6 Comparison with Other Work

Table 5 compares our method to previous TPG approaches that guarantee complete fault coverage. This includes work by Reeb et al. on deterministic generation of weights for a random pattern generator [7] (column 2 in Table 5), Akers and Jansz's test embedding [1] (column 3), Huang et al.'s Gauss-elimination-based LFSR pattern generator [11] (column 4), and Kagaris et al.'s counter-based deterministic test generator [8] (column 5). We also consider GLFSR [2] (column 6) even though it does not always guarantee complete fault coverage; it tends to require a smaller test set than many methods. Though the counter-based method (column 5) gives some of the best results, it seems inefficient for circuits whose tests are not clustered with respect to the normal numerical sequence produced by an n-bit binary counter. Omitting this column, our TPG performs best for almost all the benchmark circuits.

Next we compare the hardware overhead of the proposed TPG with that of [1] and [16]. We generated Verilog code of our most hardware-efficient implementation which consists of a binary counter (*SC*) and a layer of XOR gates (*ML*). We used the Design Analyzer software from Synopsys to obtain a fairly accurate estimate of transistor count for various ISCAS-85 benchmarks. Table 6 gives the transistor counts for various TPG designs. For columns 3 and 4, we converted the hardware cost of [16] to match our transistor count metric. For example, for c6288 which has 32 inputs, we use two $d = 16$ GLFSRs (or CAPS), each of which has XOR cost 278 (346 for CAPS). To this we add the hardware overhead of 32 D flip-flops to obtain the total transistor count of 745 (824 for CAPS). For column 2 we add the XOR counts given in [1] to the cost of the binary counter required by the benchmark circuits. Again, our TPG performs well in comparison

Benchmark circuit	Test set size					
	Weighted random patterns [7]	Akers and Jansz [1]	Multiple seeds/ polynomials [11]	Use of counters [8]	GLFSR [5]	Our TPG
c432	636	1024	320	125	n/a	256
c499	1125	1024	679	22064	n/a	256
c880	765	8192	1596	29	640	527
c1355	3059	4096	1447	122344062	1760	1024
c1908	3539	8192	3659	1169	4700	2055
c2670	7689	65536	33000	n/a	6128	6269
c3540	3351	8192	6592	970	4828	4099
c5315	2279	8192	1843	62	n/a	2048
c6288	39	512	43	98003134	n/a	128
c7552	9276	n/a	32800	n/a	n/a	24577

Table 5: Comparison of the test set size for various proposed BIST methods.

to GLFSR and CAPS for most of the benchmark circuits. It is observed that for larger circuits like c7552 and c2670 our method has higher hardware overhead. Since these circuits have large input width n, the size of subspaces required for complete fault coverage is large resulting in large numbers of basis vectors, which in turn increases the XOR logic size in our implementation.

Benchmark circuit	Akers and Jansz [1]	GLFSR	CAPS	Our TPG
c432	630	827	922	468
c499	784	926	1044	555
c880	1208	1365	1531	1099
c1355	738	926	1044	683
c1908	725	761	847	715
c2670	3668	5309	5951	9631
c3540	1027	1086	1255	1127
c5315	2708	3984	4517	3216
c6288	429	745	824	387
c7552	n/a	4617	5245	9834

Table 6: Comparison of transistor counts of the proposed TPG against other TPG methods.

7 Conclusion

We have presented a new on-chip test vector generation technique that can achieve full fault coverage at relatively low hardware cost. Our approach uses small sets of bases to generate linear vector spaces (clusters) that efficiently cover a precomputed test set. A unique feature of the proposed TPG design is its use of Gray codes to simplify the vector generation process. Experimental comparisons with a wide range of prior methods (not all of which can provide the same fault coverage) show that our approach generally provides greater test-data compression using less hardware. In many cases, the total size of the generated test set is significantly less than those produced by prior methods.

In on-going work, we are attempting to improve the clustering algorithm, which should further reduce the TPG hardware overhead. We are also extending the proposed TPG design to enable scan-based testing which will let us test sequential circuits as well.

References

[1] S. B. Akers and W. Jansz, "Test set embedding in a built-in self-test environment," *Proc. ITC*, pp. 257-263, 1989.

[2] M. Chatterjee and D. P. Pradhan, "A BIST pattern generator design for near perfect fault coverage," *IEEE Trans. Computers*, vol. 52, pp. 1543-1557, 2003.

[3] C. Dufaza and G. Cambon, "LFSR based deterministic and pseudo-random test pattern generator structures," *Proc. European Test Conf.*, pp. 27-34, 1991.

[4] D. Kagaris, "Built-in TPG with designed phaseshifts," *Proc. VLSI Test Symp.*, pp. 365-370, 2003.

[5] D. Pradhan et al., "A Hamming distance based test pattern generator with improved fault coverage," *Proc. IOLTS*, pp. 221-226, 2005.

[6] K-H Tsai et al., "Star Test: the theory and its applications," *IEEE Trans. Computer Aided Design of Integrated Circuits and Systems*, vol. 19, issue 9, pp. 1052-1064, 2000.

[7] B. Reeb and H. J. Wulderlich, "Deterministic pattern generation for weighted random pattern testing," *Proc. DATE*, pp. 30-36, 1996.

[8] D. Kagaris, S. Tragoudas and A. Majumdar, "On the use of counters for reproducing deterministic test sets," *IEEE Trans. Computers*, vol. 45, pp. 1405-1419, 1996.

[9] L. Huang et al., "Gauss-elimination-based generation of multiple seed-polynomial pairs for LFSR," *IEEE Trans. CAD*, vol. 16, pp. 1015-1024, 1997.

[10] S. Hellebrand et al., "Pattern generation for a deterministic BIST scheme," *Proc. ICCAD*, pp. 88-94, 1995.

[11] L. Huang et al., "A Gauss-elimination-based PRPG for combinational circuits," *Proc. DATE*, pp. 212-216, 1995.

[12] S. Hellebrand et al., "Built-in test for circuits with scan based on reseeding multiple-polynomial linear feedback shift registers," *IEEE Trans. Computers*, vol. 44, pp. 223-233, 1995.

[13] H. J. Wunderlich and G. Kiefer , "Bit-flipping BIST," *Proc. ICCAD*, pp. 337-343, 1996.

[14] H. K. Lee and D. S. Ha, "On the generation of test patterns for combinational circuits," Tech. Report No. 12-93, Dept. of Electrical Eng., Virginia Polytechnic Inst. and State Univ.

[15] M.L. Bushnell and V.D. Agrawal, *Essentials of Electronic Testing*, Kluwer, 2000.

[16] S. Chidambaram et al., "Comparative study of CA with phase shifters and GLFSRs," *Proc. ITC*, 2005.

[17] B. Koenemann, "Care bit density and test cube clusters: multi-level compression opportunities," *Proc. ICCD*, pp. 320-325, 2003.

[18] S. Wang et al., "XWRC: externally-loadedweighted random pattern testing for input test data compression," *Proc. ITC*, pp. 571-580, 2005.