Fully Deterministic Storage Based Logic Built-In Self-Test

Subashini Gopalsamy
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907, U.S.A.
sgopalsa@purdue.edu

Irith Pomeranz
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907, U.S.A.
pomeranz@ecn.purdue.edu

2023 IEEE 41st VLSI Test Symposium (VTS) | 979-8-3503-4630-5/23/$31.00 ©2023 IEEE | DOI: 10.1109/VTS56346.2023.10139967

***Abstract*—This paper presents a fully deterministic storage based logic built-in self-test (LBIST) approach that stores, on chip, reduced deterministic uncompressed test data sufficient for achieving complete fault coverage. The goal of this approach is to eliminate the need for pseudo-random tests, thereby reducing the test application time by reducing the number of tests required to achieve complete fault coverage. Under this approach, two types of test data are stored on chip. 1) Subsets of scan vectors obtained from a reduced set of deterministic tests, one subset per test and, 2) permutations of scan vectors stored as sets of indices, to indicate how to combine scan vectors to form tests on chip. The same permutations are applied to all the subsets, magnifying the effectiveness of each stored permutation and each subset, allowing fewer subsets as well as fewer permutations to be used. This helps in reducing the storage requirements. Experimental results are presented for single stuck-at faults in benchmark circuits and logic blocks of the OpenSPARC T1 microprocessor to demonstrate the effectiveness of this approach.**

I. Introduction

With the rapid advancement in the automotive industry and other safety critical applications, the highly integrated complex ICs used in these applications need high quality test solutions to ensure their functional safety and long-term reliability. Besides a high fault coverage, these test solutions should meet requirements such as an ability to perform in field testing and low test application time (TAT). Logic Built In Self-Test (LBIST) is a testing technique where the hardware required to test a chip is built into the chip [1]-[19]. This built-in hardware takes care of test generation, test application and response verification avoiding the need for an external tester, therefore allowing in field testing. In addition to enhancing security by removing the need for transfer of test data to and from the chip [11], LBIST also provides the ability to test at the chip frequency.

Basic LBIST techniques generate pseudo-random patterns which on their own result in low fault coverage because of the presence of random pattern resistant (RPR) faults with low detection probabilities. To increase the fault coverage, LBIST solutions based on pseudo-random patterns typically require test points to be inserted [5], [16]. A large class of methods modify pseudo-random tests to increase the fault coverage they achieve. These methods include bit-flipping [10], [13] or bit-fixing [3] and weighted random pattern generation [2]. Hybrid LBIST, another class of solutions [6], [12], [15], stores the top up deterministic patterns needed to detect RPR faults in a compressed form on a tester, and decompresses them on-chip using existing test data compression hardware [21]-[23]. Another class of solutions stores the compressed deterministic top up patterns on-chip. LFSR reseeding [7], [9] can be considered a solution of this class. With LFSR reseeding, multiple seeds for the LFSR are stored on-chip and used for test application. Stellar BIST [17] stores compressed deterministic parent patterns on-chip and decompresses them using the on-chip test data compression logic. To reduce the storage requirements, transformed derivatives of the parent patterns are obtained in [17] by complementing multiple scan slices at uniform intervals using on-chip test logic.

Another class of LBIST approaches where all the test data required to achieve complete fault coverage are stored on-chip is described in [8], [18] and [19]. This class of approaches is based on partitioning a precomputed deterministic test set into test data entries, for example scan vectors, small enough to be stored on-chip. The on-chip test generation is performed by combining the test data entries (the scan vectors) either randomly or deterministically to achieve complete fault coverage. The strength of this approach stems from the large number of tests that can potentially be formed by combining deterministic test data entries. From this large set it is possible to select a subset for detecting target faults. In [18] and [19] this property is used for achieving complete fault coverage not only for stuck-at faults but also for single-cycle gate-exhaustive faults.

In the approach described in [8], scan vectors obtained by partitioning a precomputed deterministic test set are stored separately for each scan chain. The subsets are reduced for on-chip storage. The Cartesian product of these stored subsets is performed on-chip for test generation. The number of tests generated by the Cartesian product depends on the size of each subset of scan vectors (N) and the number of subsets (n). In [8], the Cartesian product was feasible since N and n were small enough. The more recent approaches in this class do not rely on the Cartesian product to accommodate circuits for which N and n are large.

Research supported in part by NSF Grant Number CCF-2041649

In [18], a special type of pseudo-random tests is formed by pseudo-random combinations of stored deterministic test data entries (scan vectors). They are complemented by tests referred to as deterministic formed by deterministic combinations of stored scan vectors. To generate deterministic tests, additional test data are stored on-chip representing which combinations of scan vectors (indices of scan vectors in the stored set) are needed to detect the target faults. Together the two types of tests achieve complete fault coverage for single stuck-at and single-cycle gate-exhaustive faults. All the scan vectors for all the scan chains are stored in a single set in [18]. In [19], the set of scan vectors is partitioned into subsets, allowing the on-chip test generation process to focus on specific combinations of scan vectors. As a result, only pseudo-random combinations of deterministic scan vectors are used to achieve complete fault coverage in [19]. As with other types of pseudo-random tests, the number of tests needed in [18] and [19] is significantly larger than the number of deterministic tests.

Whereas [18] and [19] rely on the use of pseudo-random tests (pseudo-random combinations of stored scan vectors), the goal of this paper is to eliminate the need for pseudo-random tests. This will reduce the number of tests required to achieve complete fault coverage and allow the LBIST solution to meet the test time constraints that exist during system startup and periodic in-field testing. Thus, this paper proposes a fully deterministic storage based LBIST approach from the class of approaches described in [8], [18] and [19]. The proposed approach stores two types of test data entries on-chip. 1) Subsets of scan vectors, obtained from uncompressed deterministic tests, one subset per test. 2) Permutations of scan vector indices, stored to indicate how to combine scan vectors to form tests. The same permutations are applied to all the subsets, magnifying the effectiveness of each stored permutation and each subset, allowing fewer subsets as well as fewer permutations to be used. The permutation 0, 1, …, *n*-1, where *n* is the number of scan vectors in a subset, results in the original deterministic tests. Other permutations result in different tests referred to as deterministic derivatives. Together the original tests and their deterministic derivatives achieve complete fault coverage, eliminating the need for pseudo-random tests. Since the use of every permutation with every subset of scan vectors reduces the number of subsets as well as the number of permutations, it also reduces the storage requirements associated with input stimuli. The proposed scheme is based on the concepts from other solutions of the same class [8], [18], [19] but avoids the limitations of [8]. It adopts the underlying principles of [8], [18], [19], such as partitioning precomputed deterministic tests into scan vectors, and achieving complete fault coverage without disturbing the values of the stored deterministic test data. The permutation operation in the proposed scheme enables reuse of the stored test data entries from each subset several times in their corresponding derivatives without altering the values of the stored scan vectors.

The target faults in this paper are single stuck-at faults. The approach described in this paper can be extended to other fault models as well. Encoding [4] or test data compression [21]-[23] can be used on the original deterministic scan vectors to be stored on-chip to further reduce the on-chip storage required. Test points can be used for increasing the fault coverage without storing additional subsets. These options are not considered in this paper.

The rest of the paper is organized as follows. Section II describes the test data stored on-chip. The on-chip test generation logic is described in section III. A software procedure for computing the subsets of scan vectors and the permutations is described in Section IV. Section V describes a software procedure to reduce the number of permutations. Section VI presents the experimental results.

II. On-Chip Storage

This section describes the on-chip storage of test data entries for on-chip test application. A circuit under consideration is assumed to have *n* scan chains each of length *k*. The shorter scan chains are padded to bring their length to *k*. This is done only to simplify the discussion. As in [18], $n \approx k$ is used to make the scan vectors small enough to be stored on-chip.

The proposed scheme stores *m* subsets of scan vectors, S_0, S_1, …, S_{m-1}. Each subset consists of *n* scan vectors, corresponding to one deterministic test. Thus, $S_i = \{sv_{i,0}, sv_{i,1}, \ldots, sv_{i,n-1}\}$.

The proposed scheme also stores the permutations of scan vector indices used to construct original deterministic tests and their deterministic derivatives on-chip. Let *Y* denote the collection of *p* permutations, $Y = \{X_0, X_1, \ldots, X_{p-1}\}$. A permutation X_j is represented as a set of indices of scan vectors $X_j = (r_{j,0}, r_{j,1}, \ldots, r_{j,n-1})$. The permutation X_0 is always the original permutation (0, 1, …, *n*-1) that results in constructing the original deterministic patterns. The remaining permutations, $X_1, \ldots, X_{p-1}$, obtained by permuting the original index set, are used to construct the deterministic derivatives.

For $0 \leq i \leq m-1$ and $0 \leq j < p$, a test t_{ij} is formed by applying permutation X_j to subset S_i. The subsets of scan vectors and the permutations are chosen by a software procedure discussed in Sections IV and V.

Tables I and II illustrate the on-chip storage components and the test set generated on-chip. In this example, the circuit is assumed to have *n*=3 scan chains each of length *k*=4. Table I shows the scan vectors in two subsets S_0 and S_1. Each subset contains *n*=3 scan vectors. Table II shows the set of permutations *Y*= $\{X_0, X_1, X_2, X_3, X_4\}$ and the tests constructed from S_0 and S_1 using *Y*. Here t_{00} and t_{10} are the original deterministic tests reconstructed by applying the permutation X_0 on S_0 and S_1, respectively. The other tests t_{01}, …, t_{04} and t_{11}, …, t_{14} are the deterministic derivatives generated from S_0 and S_1, respectively, by applying the permutations X_1, X_2, X_3 and X_4 to both the subsets. Since every permutation is applied to every subset of scan vectors, the number of tests generated on-chip is *p*m*, which is 5*2 = 10 tests for the above example. Both p and m will be

TABLE I
SUBSETS OF SCAN VECTORS

i	$sv_{i,0}$	$sv_{i,1}$	$sv_{i,2}$
0	0100	1101	0010
1	1011	0000	0110

TABLE II
TEST SET CONSTRUCTED FROM SCAN VECTORS USING PERMUTATIONS

j	$r_{j,0}$	$r_{j,1}$	$r_{j,2}$	t_{0j}	t_{1j}
0	0	1	2	t_{00}={0100,1101,0010}	t_{10}={1011,0000,0110}
1	2	0	1	t_{01}={0010,0100,1101}	t_{11}={0110,1011,0000}
2	1	2	0	t_{02}={1101,0010,0100}	t_{12}={0000,0110,1011}
3	1	0	2	t_{03}={1101,0100,0010}	t_{13}={0000,1011,0110}
4	0	2	1	t_{04}={0100,0010,1101}	t_{14}={1011,0110,0000}

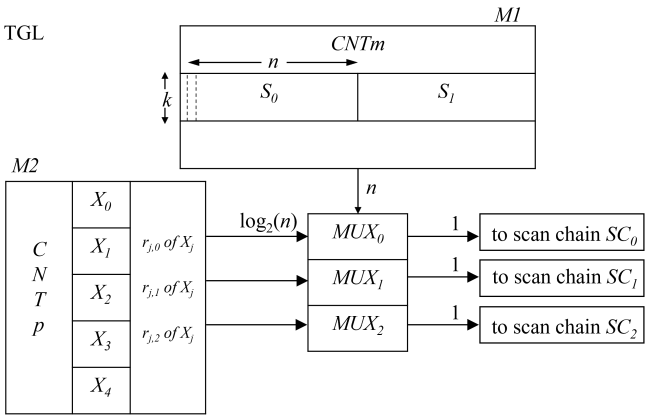

Fig. 1. On-chip Test Generation Logic.

minimized to keep the number of applied tests as small as possible.

III. ON-CHIP TEST GENERATION LOGIC

The on-chip test generation logic (TGL) for the proposed scheme is described in this section. The TGL is adopted from [19]. The TGL for the example in Tables I and II is illustrated in Figure 1. Here *M1* and *M2* are two on-chip memories storing subsets of scan vectors and permutations, respectively. These memories are dedicated to LBIST. The memory *M1* is partitioned into two blocks storing the subsets S_0 and S_1. The subset in each block has *n* = 3 scan vectors each of length *k* = 4. A single scan vector of length *k* is shown by the dashed box inside the memory block containing S_0. It takes *k* clock cycles to shift a scan vector out of the memory and into a scan chain bit by bit. The storage requirement of memory *M1* in bits is *m* * *n* * *k*. A counter denoted by *CNTm* selects a memory block thereby selecting which subset of scan vectors will be used for test application.

The memory *M2* is partitioned into *p* blocks storing *p* permutations X_0, X_1, ..., X_{p-1}. The storage requirement of memory *M2* in bits is *p* * *n* * $log_2 n$. A counter denoted by *CNTp* selects which permutation will be applied to the subset chosen by *CNTm*.

When the counter *CNTm* selects the memory block containing S_i, the *n* scan vectors of S_i are available in the output lines of the memory *M1*. From these *n* scan vectors, to select the scan vector to be shifted into the scan chain SC_q, a multiplexer MUX_q is used. MUX_q has *n* data lines routed from the output lines of memory *M1* and $log_2 n$ select lines routed from the output lines of memory *M2*. When the counter *CNTp* selects the permutation X_j, the *n* indices in X_j are available in the output of the memory *M2*. These indices $r_{j,0}$, $r_{j,1}$, ..., $r_{j,n-1}$ are routed as select inputs to the *n* multiplexers MUX_0, MUX_1, ..., MUX_{n-1}, respectively. The $log_2 n$ select inputs of a multiplexer MUX_q point to the scan vector to be loaded to the scan chain SC_q. The selected scan vector is then shifted out of the selected memory block and into scan chain SC_q bit by bit over *k* clock cycles.

During on-chip test application, every time *CNTm* is incremented, *CNTp* counts from 0 to *p*-1. This allows *p* permutations in *M2* to be applied to the subset selected by *CNTm* to produce *p* tests. The total number of tests applied to the circuit is *p*m*. The software procedure discussed in the next section computes and optimizes the number of subsets of scan vectors as well as the number of permutations to reduce the memory sizes of *M1* and *M2* and hence the overall area overhead, as well as the number of tests. The routing overhead from the TGL to the scan chains is similar to the routing overhead incurred with test data compression. In the LBIST approach, the memories *M1* and *M2* (with counters *CNTm* and *CNTp*) and *n* multiplexers replace the on-chip decompression logic. The stored test data entries can be adjusted if the scan chains are re-ordered.

For the output response, it is assumed that an output compaction logic such as a MISR [1] is used on the output side to reduce the volume of captured results.

IV. SOFTWARE PROCEDURE

This section describes a software procedure for computing the subsets of scan vectors and the permutations that reconstruct the original tests and their corresponding deterministic derivatives. This software procedure targets complete fault coverage for single stuck-at faults. The set of all the detectable stuck-at faults is denoted by F_{sa}. Undetectable faults are eliminated from consideration to simplify the procedure.

A. Overview

An overview of the procedure is shown in Figure 2. The procedure starts by initializing the undetected fault list F_{usa} = F_{sa}. In each iteration *i* = 0, 1, ..., the software procedure creates a new subset of scan vectors S_i and a set of permutations $Y_i = \{X_{i,0}, X_{i,1}, ..., X_{i,p-1}\}$. To create a new subset of scan vectors S_i in an arbitrary iteration *i*, the procedure generates a deterministic test set *T* that detects all the faults in F_{usa}. Each test in the set *T* is simulated under F_{usa}. From the set *T*, the procedure chooses the test that detects the greatest number of faults in F_{usa}. The chosen test pattern is then partitioned into *n* scan vectors each of length *k*. These *n* scan vectors are stored in a subset S_i. Next, the procedure computes a set of

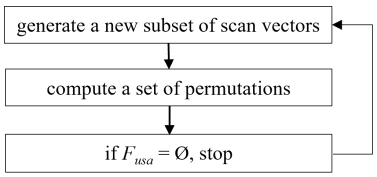

Fig. 2. Overview of software procedure.

permutations Y_i that can be applied to every existing subset of scan vectors S_0, S_1, …, S_{i-1}, S_i. The procedure computes the set $Y_i = \{X_{i,0}, X_{i,1}, \ldots, X_{i,p-1}\}$ in such a way that the deterministic derivatives produced by each permutation in Y_i detect as many faults as possible from F_{usa}. This property of the permutations magnifies their effectiveness thereby allowing fewer subsets as well as fewer permutations to be used.

At the end of the permutation computation step, the procedure generates a test set T_i. The tests in T_i are constructed by applying every permutation in Y_i to every existing subset of scan vectors S_0, S_1, …, S_{i-1}, S_i. Every test in T_i is then fault simulated with fault dropping under F_{sa}. After fault simulation, the undetected fault list F_{usa} is updated. The procedure terminates when F_{usa} is empty which implies that all the faults from F_{sa} are detected. The steps involved in the permutation computation are described next.

B. Permutation Computation

Figure 3 shows the steps involved in the computation of a set of permutations. In an arbitrary iteration *i*, the software procedure computes a set of permutations $Y_i = \{X_{i,0}, X_{i,1}, \ldots, X_{i,p-1}\}$. The first permutation of Y_i is the original permutation $X_{i,0}$ = (0, 1, …, *n*-1). The rest of the permutations $X_{i,1}$, …, $X_{i,p-1}$ are selected randomly to avoid the complexity of forming permutations using a deterministic procedure. The procedure produces a random permutation by randomly permuting the original index set (0, 1, …, *n*-1). It checks for every random permutation whether the deterministic derivatives obtained from its application on existing subsets of scan vectors detect any faults from F_{usa}. It then adds only permutations that detect new faults to the set Y_i. The procedure stops producing the permutations when the last *w* random permutations do not increase the fault coverage.

The value of *w* should be small enough to avoid long run times, and large enough to find as many useful permutations as possible. For the experimental results reported in this paper, *w* is set to 15000 for all the circuits. This value was selected by experimenting with different values and attempting to balance the run time and the quality of the results.

C. Removing Unnecessary Permutations

The set Y_i may contain permutations that are not necessary for the fault coverage attained by S_0, S_1, …, S_{i-1}, S_i and Y_i. For example, the random permutation $X_{i,1}$ may not be necessary after adding the rest of the permutations $X_{i,2}$, …, $X_{i,p-1}$. The original permutation $X_{i,0}$ is excluded from this analysis since the original tests are typically necessary for achieving complete fault coverage.

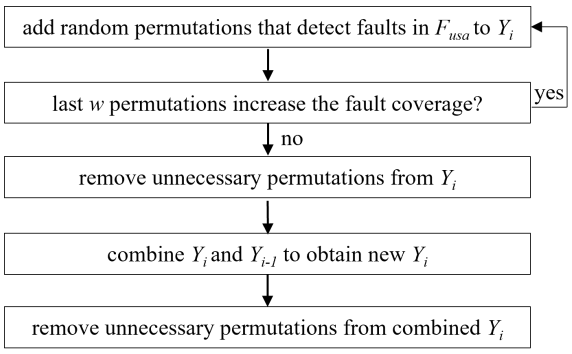

Fig. 3. Steps in permutation computation.

The software procedure removes the unnecessary permutations as described next. The concepts are adopted from [20]. The procedure constructs test sets $T_{i,1}$, …, $T_{i,p-1}$ for the permutations $X_{i,1}$, …, $X_{i,p-1}$, respectively. A test set $T_{i,j}$ contains all the deterministic derivatives constructed from applying the permutation $X_{i,j}$ to every existing subset S_0, S_1, …, S_{i-1}, S_i. The procedure considers the test sets in different orders to remove unnecessary test sets and their corresponding permutations. Let T_i represent the collection of all the test sets. First the procedure considers the test sets in T_i in the original order followed by a random order and finally the reverse order. While considering the test sets in a particular order, the procedure simulates the ordered test sets in T_i under F_{sa} with fault dropping. It then associates with every test set $T_{i,j}$ the number of faults it detects. This number is denoted by $f(T_{i,j})$. The procedure rearranges the test sets by decreasing order of $f(T_{i,j})$ and simulates them again. If the number of faults detected by a test set becomes zero, the test set and the corresponding permutation are removed from T_i and Y_i, respectively. The procedure repeats the rearranging and fault simulation three times under each initial order. The sets T_i and Y_i are updated every time. The procedure reinitializes F_{sa} to include all the target faults before changing the order.

D. Combining Sets of Permutations

Although Y_i targets the subsets S_0, S_1, …, S_i directly, it is possible that permutations included in Y_{i-1}, targeting S_0, S_1, …, S_{i-1}, would be useful after S_i is added. It is important to take advantage of effective permutations from Y_{i-1} since this will reduce the number of iterations, and the number of subsets. For this purpose, the procedure combines Y_i with Y_{i-1} as follows. The procedure computes the average number of faults $f_a(T_i)$ detected by a subset $T_{i,j} \in T_i$. It also computes the average $f_a(T_{i-1})$, where T_{i-1} represents the collection of test sets from the previous iteration. The procedure then combines the permutations from Y_i and Y_{i-1} in the following fashion. Every $X_{i,j}$ with $f(T_{i,j}) \geq f_a(T_i)$ followed by every $X_{i-1,j}$ with $f(T_{i-1,j}) \geq f_a(T_{i-1})$ followed by $X_{i,j}$ with $f(T_{i,j}) < f_a(T_i)$ followed by $X_{i-1,j}$ with $f(T_{i-1,j}) < f_a(T_{i-1})$. This order ensures that the permutations that detect the largest numbers of faults, from both the sets Y_i and Y_{i-1}, are preserved. Unnecessary permutations are

removed from the combined set using the same procedure described earlier, thereby allowing fewer permutations at the end of each iteration. The faults that remain undetected at the end of iteration *i* are assigned to F_{usa}.

V. Improving the Permutations

The improvement procedure described in this section considers the permutations one by one for improvement. The order of the permutations ensures that permutations detecting more faults are considered earlier, making it more likely that permutations at the end of the set *Y* will become unnecessary. The procedure adds the improved permutations to a set Y_{mod}. If F_{mod}, which is a set of all the faults detected by the deterministic derivatives, is not empty after considering all the permutations in *Y* for improvement, the procedure repeats the same process for as many passes as needed, until F_{mod} becomes empty. The entire procedure is repeated for several iterations where at the end of each iteration Y_{mod} is assigned to *Y*. Each iteration tries to improve the set *Y* obtained from the previous iteration. The procedure stops when *M* consecutive iterations do not improve any permutation or reduce the number of permutations. For the experimental results reported in this paper, *M* is set to 4 for all the circuits.

The steps involved in an arbitrary iteration are shown in Figure 4. In an arbitrary iteration, the procedure attempts to improve the permutations X_1, X_2, ..., X_{p-1} from *Y*. When X_j is considered, the procedure constructs the test set T_j by applying X_j to every subset of scan vectors S_0, S_1, ..., S_{m-1}. It then simulates T_j under F_{mod} to compute the number of faults T_j detects, denoted by $f(T_j)$. It is important to simulate T_j because it is possible that the permutations that were modified before X_j detect the faults that were originally detected by X_j. If $f(T_j) = 0$, no permutation based on X_j is added to Y_{mod}. If $f(T_j) > 0$, the improvement procedure attempts to modify the permutation $X_j = (r_{j,0}, r_{j,1}, \ldots, r_{j,n-1})$ as follows. It considers the indices $r_{j,0}, r_{j,1}, \ldots, r_{j,n-1}$ one by one. It replaces the index under consideration with every other index from the set $\{0, 1, \ldots, n\text{-}1\}$. For every index in X_j, there are *n*-1 options available for replacement. The total number of modifications performed on X_j is *n* * (*n*-1). The procedure considers these options one at a time. For every replacement option for the index $r_{j,m}$ where $0 \leq m \leq n-1$, a modified permutation X_j^{mod} and its corresponding test set T_j^{mod} are obtained. The procedure simulates T_j^{mod} under F_{mod} and performs the following check. If the modified permutation X_j^{mod} detects more faults than its previous version, the replacement option is accepted, and the procedure continues to modify X_j^{mod} to improve it further. If not, the index under consideration is restored to its previous value. After exploring all the replacement options, the improved permutation X_j^{mod} is added to Y_{mod} and the faults detected by T_j^{mod} are dropped from F_{mod}.

VI. Experimental Results

The software procedure that computes the subsets of scan vectors S_0, S_1, ..., S_{m-1} and set of permutations *Y* was applied to single stuck-at faults in ISCAS-89, ITC-99 and IWLS-05 benchmark circuits, and three logic blocks (*ffu*, *spu* and *exu*) of the OpenSPARC T1 microprocessor [24]. Consideration of these logic blocks demonstrates the applicability of the proposed approach to logic blocks of a large processor. A commercial tool was used for test generation, fault simulation and reordering. The experimental results are shown in Table III. Table III also compares the proposed method with [19] and a pseudo-random LBIST method available in the same commercial tool used for the implementation of the software procedure.

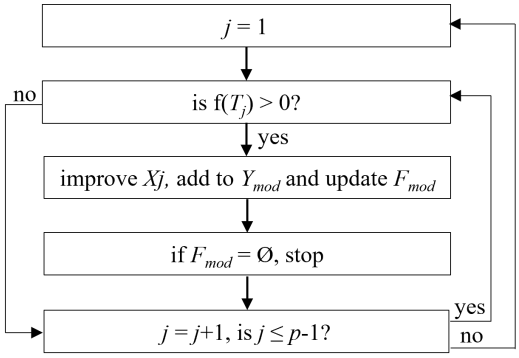

Fig. 4. Steps in an arbitrary iteration for improving the permutations.

In Table III, after the circuit name, column *n* shows the number of scan chains which is equal to the number of scan vectors in each subset S_i. Column *k* shows the length of each scan chain. Column *ATPG tests* shows the number of uncompressed deterministic tests that achieves complete fault coverage. Column *m* shows the number of subsets of scan vectors. Column *p* shows the number of permutations in the set *Y*. Column *p*m* shows the total number of tests generated on-chip to achieve complete fault coverage. Column *bits* shows the number of storage bits required for *m* subsets of scan vectors (subcolumn *S*), number of bits required for storing *p* permutations (subcolumn *Y*) and total number of bits for storing both the subsets and permutations (subcolumn *tot*). Column *red%* shows the storage reduction percentage, computed as (*ATPG bits-tot bits*)/*ATPG bits*. Here *ATPG bits* is the number of storage bits required for ATPG tests. The ATPG test set is generated by the same commercial ATPG tool used for implementing the software procedure. Column *FC%* shows the complete fault coverage achieved by the proposed LBIST scheme. Column *rt* shows the run time in minutes. This is the total run time taken to execute the entire software procedure. Along with the commercial tool, the implementation uses python scripts which are slow.

Column [19] and column *random tests* present the results from [19] and a pseudo-random LBIST method, respectively, to compare with the proposed LBIST scheme. Column [19] subcolumn *tests* shows the number of test patterns generated by [19], on chip, to achieve complete stuck-at fault coverage. Subcolumn *bits* shows the number of bits required for storing the scan vectors used in [19]. Subcolumn *red%* shows the

TABLE III
EXPERIMENTAL RESULTS

circuit	n	k	ATPG tests	m	p	p*m	bits S	bits Y	bits tot	red%	FC%	rt	[19] tests	[19] bits	[19] red%	random tests
s35932	42	43	59	1	23	23	1806	5796	7602	92.71	100.0	246	-	-	-	128
sasc	12	12	35	4	11	44	576	528	1104	76.46	100.0	13	384	144	95.04	256
systemcdes	18	18	104	3	31	93	972	2790	3762	88.84	99.74	38	768	216	99.15	704
des_area	18	18	195	5	30	150	1620	2700	4320	92.85	100.0	41	1152	720	98.34	768
usb_phy	11	11	39	4	14	56	484	616	1100	75.47	100.0	12	1024	176	95.09	1728
aes_core	28	29	376	2	253	506	1624	35420	37044	87.55	100.0	2580	-	-	-	2688
systemcaes	31	31	208	13	28	364	12493	4340	16833	91.32	100.0	546	4096	992	99.12	13504
b04	9	9	78	5	25	125	405	900	1305	79.34	96.91	18	49152	108	96.85	20032
s1423	10	10	70	7	17	119	700	680	1380	79.25	100.0	35	8192	320	86.48	25856
b05	6	7	90	21	17	357	882	306	1188	66.15	100.0	16	20480	120	94.54	34432
s5378	14	15	129	11	48	528	2310	2688	4998	80.72	99.89	93	16384	1920	91.03	49344
s13207	16	16	60	13	14	182	3328	896	4224	71.61	95.75	78	53248	5616	96.59	213056
wb_conmax	44	44	178	8	68	544	15488	17952	33440	90.12	99.96	3840	-	-	-	310976
s9234	13	14	98	7	36	252	1274	1872	3146	81.34	99.17	144	-	-	-	406144
ffu	38	38	192	6	68	408	8664	15504	24168	91.17	97.73	1860	-	-	-	508224
b07	7	7	71	19	11	209	931	231	1162	66.60	100.0	15	98304	192	93.03	*1M
b14	16	16	284	63	51	3213	16128	3264	19392	72.80	99.91	4422	-	-	-	*1M
b15	21	22	459	73	125	9125	33726	13125	46851	77.62	99.52	12696	-	-	-	*1M
b20	22	22	332	69	58	4002	33396	6380	39776	74.29	99.92	3660	-	-	-	*1M
s38417	39	39	149	19	122	2318	28899	28548	57447	74.19	99.98	8094	-	-	-	*1M
s38584	34	35	127	17	37	629	20230	7548	27778	81.38	99.70	660	-	-	-	*1M
DMA	49	49	351	102	115	11730	244902	33810	278712	66.86	99.64	26849	-	-	-	*1M
i2c	12	13	73	24	15	360	3744	720	4464	58.96	99.91	86	24576	624	90.44	*1M
pci_spoci_ctrl	9	10	174	55	31	1705	4950	1116	6066	59.93	100.0	384	425984	1040	91.42	*1M
simple_spi	12	13	62	11	18	198	1716	864	2580	71.88	100.0	23	28672	364	93.07	*1M
spi	17	17	500	18	72	1296	5202	6120	11322	91.86	99.93	225	32768	1088	99.02	*1M
tv80	19	20	466	76	102	7752	28880	9690	38570	77.93	100.0	3240	-	-	-	*1M
spu	46	46	198	114	28	3192	241224	7728	248952	40.27	99.04	13380	-	-	-	*1M
exu	54	55	421	151	75	11325	448470	24300	472770	61.93	98.37	45372	-	-	-	*1M

storage reduction achieved in [19]. It is to be noted that the circuits are synthesized differently and the test sets are different in [19]. Column *random tests* shows the number of pseudo-random test patterns generated by a pseudo-random LBIST method to achieve complete fault coverage. An asterisk indicates that the pseudo-random test generation is stopped at 1M tests without achieving complete fault coverage. The circuits in Table III are arranged by increasing order of number of random tests.

The following points can be seen from Table III. The proposed fully deterministic LBIST scheme achieves complete fault coverage for all the circuits, thereby eliminating the need for pseudo-random tests. This in turn reduces the number of tests. Compared with [19] and a pseudo-random LBIST method, the proposed scheme uses a significantly smaller number of tests. This is the main purpose of the work. For example, in the case of *spi*, the reduction in number of tests is 32768/1296=25.28.

The software procedure reduces the storage requirements for all the circuits considered. The storage reduction achieved by the proposed scheme is lesser than [19] which is inevitable when no random tests are used.

The improvement software procedure helps in reducing the initial number of permutations by over 52%. This procedure is important for further reducing the storage requirement and thus the hardware overhead.

The circuit *spu* has the lowest storage reduction of 40%. Without the last 1% fault coverage, the storage reduction for *spu* is 92%. Similarly, for *i2c*, the reduction is 82% without the last 2% fault coverage. This points to the possibility that test points will be useful in keeping the storage reduction high.

VII. CONCLUSION

This paper described a fully deterministic storage based logic built-in self-test (LBIST) approach that stores, on chip, reduced deterministic uncompressed test data sufficient for achieving complete fault coverage. This approach eliminated the need for pseudo-random tests, thereby reducing the test application time by reducing the number of tests required to achieve complete fault coverage. Under this approach, two types of test data are stored on chip. 1) Subsets of scan vectors obtained from a reduced set of deterministic tests, one subset per test and, 2) permutations of scan vectors stored as sets of indices, to indicate how to combine scan vectors to form tests on chip. The same permutations are applied to all the subsets, magnifying the effectiveness of each stored permutation and each subset, allowing fewer subsets as well as fewer permutations to be used. This helps in reducing the storage requirements. Experimental results for single stuck-at faults in benchmark circuits and logic blocks of the OpenSPARC T1 microprocessor demonstrated the effectiveness of this approach.

REFERENCES

[1] P. H. Bardell, W. H. McAnney and J. Savir, Built-In Test for VLSI Pseudorandom Techniques, Wiley Interscience, 1987.

[2] R. Kapur, S. Patil, T. J. Snethen and T. W. Williams, "Design of an efficient weighted random pattern generation system," *Proc. ITC*, 1994, pp. 491-500.

[3] N. A. Touba and E. J. McCluskey, "Altering a pseudo-random bit sequence for scan-based BIST," in *Proc. ITC*, 1996, pp. 167-175.

[4] V. Iyengar, K. Chakrabarty and B. T. Murray, "Built-in self testing of sequential circuits using precomputed test sets," in *Proc. VTS*, 1998, pp. 418-423.

[5] M. Nakao, S. Kobayashi, K. Hatayama, K. Iijima and S. Terada, "Low overhead test point insertion for scan-based BIST," *Proc. ITC*, 1999, pp. 348-357.

[6] D. Das and N. A. Touba, "Reducing test data volume using external/LBIST hybrid test patterns," in *Proc. ITC*, 2000, pp. 115-122.

[7] H. G. Liang, S. Hellebrand and H. -J. Wunderlich, "Two-dimensional test data compression for scan-based deterministic BIST," in *Proc. ITC*, 2001, pp. 894-902.

[8] I. Pomeranz and S. M. Reddy, "A partitioning and storage based built-in test pattern generation method for scan circuits," in *Proc. VLSI Design Conf.*, 2002, pp. 677-682.

[9] A. A. Al-Yamani, S. Mitra and E. J. McCluskey, "BIST reseeding with very few seeds," in *Proc. VTS*, 2003, pp. 69-74.

[10] V. Gherman, H. -J. Wunderlich, H. Vranken, F. Hapke, M. Wittke and M. Garbers, "Efficient pattern mapping for deterministic logic BIST," in *Proc. ITC*, 2004, pp. 48-56.

[11] S. Pateras, "Security vs. test quality: fully embedded test approaches are the key to having both," in *Proc. Intl. Test Conf.*, 2004, pp. 1413.

[12] A. Jas, C. V. Krishna and N. A. Touba, "Weighted pseudorandom hybrid BIST," in *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, Dec. 2004, Vol. 12, No. 12, pp. 1277-1283.

[13] V. Gherman, H. -J. Wunderlich, J. Schloeffel and M. Garbers, "Deterministic Logic BIST for Transition Fault Testing," in *Proc. ETS*, 2006, pp. 123-130.

[14] R. S. Oliveira, J. Semião, I. C. Teixeira, M. B. Santos and J. P. Teixeira, "On-line BIST for performance failure prediction under aging effects in automotive safety-critical applications," in *Proc.* Latin American Test Workshop, 2011, pp. 1-6.

[15] M. Filipek, G. Mrugalski, N. Mukherjee, B. Nadeau-Dostie, J. Rajski, J. Solecki and J. Tyszer, "Low-Power Programmable PRPG With Test Compression Capabilities", in *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, June 2015, Vol. 23, No. 6, pp. 1063-1076.

[16] E. Moghaddam, N. Mukherjee, J. Rajski, J. Tyszer and J. Zawada, "Test point insertion in hybrid test compression/LBIST architectures," *Proc. ITC*, 2016, pp. 1-10.

[17] Y. Liu, N. Mukherjee, J. Rajski, S. M. Reddy and J. Tyszer, "Deterministic Stellar BIST for In-System Automotive Test," in *Proc. ITC*, 2018, pp. 1-9.

[18] I. Pomeranz, "Storage-Based Built-In Self-Test for Gate-Exhaustive Faults," in *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, Oct. 2021, Vol. 40, No. 10, pp. 2189-2193.

[19] I. Pomeranz, "Zoom-In Feature for Storage-Based Logic Built-In Self-Test," in *Proc. Symp. Defect Fault Tolerance*, 2021, pp. 1-6.

[20] X. Lin, J. Rajski, I. Pomeranz and S. M. Reddy, "On static test compaction and test pattern ordering for scan designs," *Proc. ITC*, 2001, pp. 1088-1097.

[21] B. Koenemann, "Care bit density and test cube clusters: multi-level compression opportunities," *Proc.* International Conference on Computer Design, 2003, pp. 320-325.

[22] J. Rajski, J. Tyszer, M. Kassab and N. Mukherjee, "Embedded deterministic test," in *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, May 2004, Vol. 23, No. 5, pp. 776-792.

[23] P. Wohl, J. A. Waicukauski, S. Patel, F. DaSilva, T. W. Williams and R. Kapur, "Efficient compression of deterministic patterns into multiple PRPG seeds," *Proc. ITC*, 2005, pp. 10 pp.-925.

[24] Oracle 2022, *OpenSPARC T1 Specifications*, https://www.oracle.com/servers/technologies/opensparc-overview.html.