Direct Computation of $LFSR$-Based Stored Tests for Broadside and Skewed-Load Tests

Irith Pomeranz

***Abstract*— Using both broadside and skewed-load tests for delay faults provides a higher fault coverage and more compacted test sets. An earlier work showed that it is possible to share input test data between broadside and skewed-load tests, and thus reduce the input test data volume. This paper develops an algorithm for computing stored tests that can be used for applying both broadside and skewed-load tests in the context of a specific test data compression method. Under this method, a programmable linear-feedback shift-register is used for on-chip decompression. In the stored test data, a stored test consists of a seed and two primary input vectors. The seed determines the scan-in state of the applied broadside or skewed-load test as well as the additional scan-in vector required for a skewed-load test. In the algorithm developed in this paper, stored tests are computed directly without first computing broadside or skewed-load tests. This avoids situations where the tests cannot be compressed or do not have common input test data.**

***Index Terms*— Broadside tests, linear-feedback shift-register ($LFSR$), skewed-load tests, test data compression, test generation, transition faults.**

I. Introduction

Test data compression methods reduce the test data volume by storing compressed tests and compacted output responses [1]-[19]. On-chip, decompression logic is used for loading the input test data into the circuit. Output compaction logic is used for capturing the output response.

When the same input test data is used for applying several different tests, the input test data volume is reduced further [9], [10], [12], [14], [15], [16], [18]. In [12], a stored test consists of a scan-in state, two primary input vectors (for two clock cycles of a test), and two additional scan-in vectors. These are used for applying one broadside (launch-on-capture) and two skewed-load (launch-on-shift) tests. The skewed-load tests differ in the scan-in vector used in the scan shift cycle that follows the scan-in operation. The advantages of using both types of tests are: (1) it is possible to achieve a higher delay fault coverage since some delay faults are only detected by broadside tests while others are only detected by skewed-load tests; and (2) it is possible to achieve higher levels of test compaction by selecting the test type that leads to the detection of more faults [20]-[21]. The number of stored tests for the application of broadside and skewed-load tests is reduced in [12] by applying three tests based on every stored test.

The approach in [12] is developed without considering a specific test data compression method. It starts from a test set that consists of broadside and skewed-load tests where the two test types are generated separately. It extracts from the test set triples that consist of scan-in states, pairs of primary input vectors, and additional scan-in vectors for skewed-load tests (two are used in [12]). Each triple is then associated with three tests, a broadside test and two skewed-load tests. Unnecessary triples are removed, and remaining triples are modified to increase their contribution to the fault coverage. This allows additional triples to be removed.

A more effective algorithm for the computation of common input test data for broadside and skewed-load tests has the characteristics discussed next. Such an algorithm is developed in this paper.

(1) The algorithm considers a specific method for test data compression. In this paper, a linear-feedback shift-register ($LFSR$) is used as the on-chip decompression logic, and stored test data includes seeds for the $LFSR$. A seed is associated with the pair of primary input vectors required for applying a test. The same seed is also used for producing the additional scan-in vector required for a skewed-load test. Thus, a stored test consists of a seed and a pair of primary input vectors. One broadside and one skewed-load test are applied based on every stored test. Scan-in vectors for skewed-load tests are not stored. A programmable $LFSR$ is used for a higher fault coverage [2], [13].

(2) Existing test generation procedures do not generate common input test data for broadside and skewed-load tests. Instead, they generate the two types of tests separately. Even if a test set that consists of broadside and skewed-load tests is generated for a specific test data compression method, it does not utilize the possibility of sharing input test data between the two test types. Moreover, if scan-in states are modified as in [12] to allow sharing of input test data between broadside and skewed-load tests, it is necessary to check that the modified scan-in states can be compressed by the test data compression method. To address these issues, the procedure described in this paper generates stored tests directly. In particular, it generates seeds for the $LFSR$ instead of generating scan-in states. Every stored test s_j is associated with two applied tests, a broadside test b_j and a skewed-load test w_j. The contribution of s_j to the fault coverage consists of the combined contribution of b_j and w_j. The procedure generates s_j directly, and evaluates it based on the number of faults detected by b_j and w_j.

The procedure for constructing a stored test set consists of two subprocedures. The first subprocedure generates an initial stored test set randomly. A stored test in the initial test set is constructed by randomly specifying a seed and two primary input vectors. This provides an initial fault coverage and an initial level of sharing of input test data between broadside

Irith Pomeranz is with the School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907, U.S.A. (e-mail: pomeranz@ecn.purdue.edu).

This work was supported in part by NSF grant CCF-1714147

and skewed-load tests. As an alternative, the initial stored test set can be computed by any test generation procedure for broadside and skewed-load tests that accommodates the constraints of the test data compression method.

The second subprocedure modifies stored tests in order to increase the contribution of each one to the fault coverage. When a stored test s_j is modified, the goal is to increase the combined contribution of the broadside test b_j and the skewed-load test w_j to the fault coverage. The modification is carried out by complementing bits of s_j one by one. Bit complementation was shown to be effective for test compaction in [22]. It is also used for the input test data in [12]. In [17] and [19], an $LFSR$-based test generation procedure modifies seeds by complementing their bits one by one in order to achieve diagnosis objectives or detect path delay faults. In this paper, this approach is applied to stored tests that include $LFSR$ seeds and pairs of primary input vectors in order to increase the combined contribution to the fault coverage achieved by the broadside and skewed-load tests obtained from every stored test. Modifying seeds (as in [17] and [19]) instead of scan-in states (as in [12] and [22]) has a significant advantage in computational effort since seeds have significantly fewer bits than scan-in states.

The second subprocedure is applied iteratively using existing and new random stored tests. By modifying stored tests to increase their contribution to the fault coverage, the procedure increases the fault coverage of the applied test set. It also reduces the number of stored tests, thus increasing the sharing of input test data between broadside and skewed-load tests, and reducing the input test data volume. Finally, the procedure is applied selectively in an attempt to eliminate from the set of stored tests the ones that detect single faults. This reduces the number of stored tests further, reducing the input test data volume and increasing the sharing of input test data further.

In general in the context of test data compression, the second subprocedure modifies the stored tests computed for a particular approach to test data compression. As stored tests are modified, applied tests are computed and simulated to check the effects of the modification. As a special case, this approach is applicable when the initial test set is computed by solving linear equations for a linear decompression logic. Existing test generation procedures do not guarantee sharing of test data between broadside and skewed-load tests, and the second subprocedure contributes to such sharing. The second subprocedure can also be applied to the stored tests when different approaches are used for producing more than one applied test from every stored test as in [9], [10], [12], [14], [15], [16] or [18].

The paper is organized as follows. The format of a stored test and the resulting applied tests are described in Section II. A procedure for generating a stored test set where every stored test is used for applying a broadside and skewed-load test is described in Section III. Experimental results are presented in Section IV.

II. Stored and Applied Tests

This section describes the format of a stored test and how the applied broadside and skewed-load tests are obtained.

TABLE I
Example Tests

j	test	$p_{j,0}$	$v_{j,0}$	$p^b_{j,1}/p^w_{j,1}$	$v_{j,1}$
0	b_0	01111010110010	001	11111010110000	100
0	w_0	01111010110010	001	00111101011001	100
1	b_1	01011001000111	100	00000001000100	101
1	w_1	01011001000111	100	10101100100011	101

A stored test is denoted by $s_j = \langle \sigma_j, v_{j,0}, v_{j,1} \rangle$. It consists of a seed σ_j for the $LFSR$, and two primary input vectors, $v_{j,0}$ and $v_{j,1}$. Suppose that the longest scan chain has K state variables. When the seed σ_j is loaded into the $LFSR$, and the $LFSR$ is clocked for K clock cycles, it produces a scan-in state denoted by $p_{j,0}$. After an additional clock cycle, the $LFSR$ produces the scan-in vector q_j.

The stored test s_j is used for applying two tests, a broadside test and a skewed-load test. The broadside test is denoted by $b_j = \langle p_{j,0}, v_{j,0}; p^b_{j,1}, v_{j,1} \rangle$. Here, $p_{j,0}$ and $v_{j,0}$ are the present-state and primary input vector for the first clock cycle of the test; and $p^b_{j,1}$ and $v_{j,1}$ are the present-state and primary input vector for the second clock cycle. After $p_{j,0}$ is scanned in, the two primary input vectors $v_{j,0}$ and $v_{j,1}$ are applied in two consecutive functional capture cycles. The state $p^b_{j,1}$ is the next-state obtained for $p_{j,0}$ and $v_{j,0}$ during the first functional capture cycle. This is also the present-state for the second functional capture cycle.

The skewed-load test is denoted by $w_j = \langle p_{j,0}, v_{j,0}; p^w_{j,1}, v_{j,1} \rangle$. After $p_{j,0}$ is scanned in, the primary input vector $v_{j,0}$ is applied in scan shift mode with scan-in vector q_j produced by the $LFSR$. The state $p^w_{j,1}$ is the resulting next-state. The second clock cycle of the test is a functional capture cycle where the primary input vector $v_{j,1}$ is applied with the circuit in state $p^w_{j,1}$.

For illustration, ISCAS-89 benchmark circuit $s298$ is considered next. The circuit has three primary inputs, and 14 state variables that are included in a single scan chain. The scan chain is shifted to the right. The 4-bit primitive $LFSR$ from [23] is used for this example.

Two stored tests are considered, $s_0 = \langle 0010, 001, 100 \rangle$ and $s_1 = \langle 1111, 100, 101 \rangle$. The resulting broadside and skewed-load tests are shown in Table I. When $\sigma_0 = 0010$ is loaded into the $LFSR$, and the $LFSR$ is clocked for 14 clock cycles, the scan-in state $p_{0,0} = 01111010110010$ is obtained. With an additional clock cycle, the $LFSR$ produces the scan-in vector $q_0 = 0$. This yields the broadside and skewed-load tests b_0 and w_0 shown in Table I.

For s_1, the scan-in state is $p_{1,0} = 01011001000111$ and the additional scan-in vector is $q_1 = 1$. This yields the broadside and skewed-load tests b_1 and w_1 shown in Table I.

Table I demonstrates that the input values to the combinational logic during the first clock cycles of b_j and w_j are the same. The tests differ significantly in the second clock cycle. This allows them to detect different faults.

Although the two test types are stored using common test data, it is not necessary to alternate between them during test application. Instead, it is possible to first apply all the broadside tests, and then apply all the skewed-load tests (or vice versa).

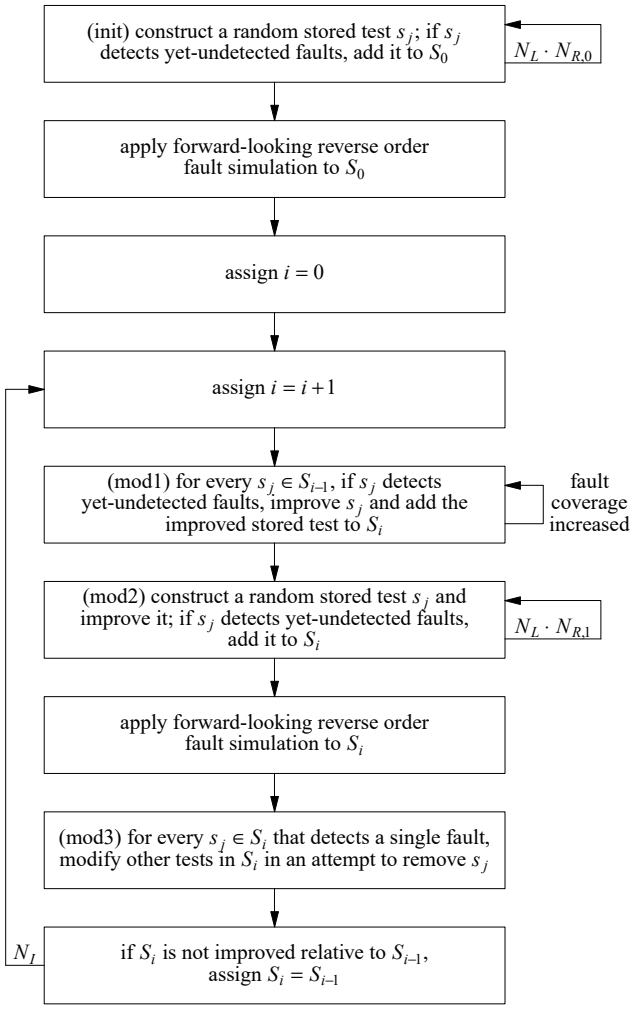

Fig. 1. Procedure for generating a stored test set

III. Generating a Stored Test Set

This section describes a procedure for generating a stored test set where every stored test is used for applying a broadside and skewed-load test. The procedure attempts to ensure that both tests together increase the fault coverage as much as possible. However, if only one of the tests increases the fault coverage, it is possible to avoid the application of the other test.

The procedure considers a given set L of $LFSR$s. A set of $LFSR$s can be implemented by a programmable $LFSR$ with minimal additional logic compared with a single $LFSR$ [2]. The use of multiple $LFSR$s increases the achievable fault coverage and makes it easier to compress tests [2], [13]. The $LFSR$s in L are assumed to have distinct numbers of bits, and an $LFSR$ is described by its number of bits. Consequently, we have that $L = \{l_0, l_1, ..., l_{N_L-1}\}$, with $l_0 < l_1 < ... < l_{N_L-1}$. As a special case it is possible to consider a single $LFSR$ by using $N_L = 1$, and $L = \{l_0\}$.

The overall flow of the procedure for computing a stored test set is illustrated by Figure 1. The next subsections describe the generation of an initial set of stored tests, and the iterative procedure that modifies stored tests in order to reduce their number and increase the fault coverage. The initial set of stored tests is denoted by S_0. The iterative procedure produces sets of stored tests denoted by S_1, S_2, ...

Any one of the sets of stored tests that the procedure produces may contain tests that are not necessary for achieving the fault coverage. The procedure eliminates unnecessary stored tests by applying forward-looking reverse order fault simulation. During this process it considers the broadside test b_j and the skewed-load test w_j together for every stored test s_j. The procedure removes s_j from the stored test set if neither b_j nor w_j is necessary for detecting any target faults.

A. Initial Set of Stored Tests

The initial set of stored tests is denoted by S_0. Its construction is referred to as $init$ in Figure 1. Initially, $S_0 = \emptyset$, and F contains all the target faults (transition faults in this paper). For every number of $LFSR$ bits $l_k \in L$, the procedure generates $N_{R,0}$ stored tests randomly, where $N_{R,0}$ is a constant. To construct a stored test $s_j = \langle \sigma_j, v_{j,0}, v_{j,1} \rangle$, the procedure specifies the bits of σ_j, $v_{j,0}$ and $v_{j,1}$ randomly. It computes the broadside and skewed-load tests b_j and w_j obtained from s_j. It then simulates F under b_j and w_j. If any fault is detected by either b_j or w_j, the procedure adds s_j to S_0, and removes the detected faults from F. Otherwise, s_j is discarded.

In the case of $s298$ with $L = \{4, 5\}$ and $N_{R,0} = 10$, the procedure includes a total of 15 stored tests in S_0. Forward-looking reverse order fault simulation reduces the number of stored tests in S_0 to 13. Of these tests, seven use the 4-bit and six use the 5-bit $LFSR$. The transition fault coverage of the initial test set is 63.93%.

B. Improving the Set of Stored Tests

For $i > 0$, an improved set of stored tests S_i is first computed by modifying stored tests from S_{i-1}. This is referred to as $mod1$ in Figure 1. The goal of the modification is to reduce the number of stored tests and increase the fault coverage by increasing the number of faults that each stored test detects. A larger increase is likely to occur when both the broadside and skewed-load test obtained from a stored test contribute to the fault coverage. Thus, the sharing of input test data between the two test types is improved by the modification.

The procedure also modifies additional randomly constructed stored tests. This is referred to as $mod2$ in Figure 1.

Once a new set of stored tests S_i is obtained, the procedure attempts to eliminate the stored tests in S_i that detect single faults. This is referred to as $mod3$ in Figure 1. For every stored test $s_j \in S_i$ that detects a single fault, the procedure attempts to modify every other test in S_i so as to detect the fault that s_j detects. When this is successful, s_j can be removed, and no additional stored tests are considered for s_j.

The three parts of the procedure, where it considers existing and new random stored tests, and attempts to remove tests that detect single faults, are described below.

The final set S_i is considered improved compared with S_{i-1} if its fault coverage is higher than that of S_{i-1}, or the number

of bits required for storing it is lower than that of S_{i-1}. If S_i is not improved relative to S_{i-1}, the procedure assigns $S_i = S_{i-1}$. It then uses $S_i = S_{i-1}$ for computing S_{i+1}. The procedure will obtain $S_{i+1} \neq S_i$ because of the random decisions made during the construction of a new set of stored tests.

The procedure terminates after N_I iterations where the test set is not improved, for a constant N_I.

C. Using Existing Stored Tests

This subsection describes the first part of the modification procedure ($mod1$ in Figure 1).

The modification procedure initially assigns $S_i = \emptyset$, and includes all the target faults in the set F. It also reorders S_{i-1} such that stored tests appear by order of ascending number of $LFSR$ bits. This is important for ensuring that S_i uses $LFSR$s with the smallest possible numbers of bits. Among the stored tests with the same number of $LFSR$ bits, the tests are ordered by descending number of detected faults. This ensures that more effective tests will be considered earlier for contributing new tests to S_i.

To construct S_i, the procedure considers every stored test $s_j \in S_{i-1}$. It computes the tests b_j and w_j, and simulates F under both tests. Let the number of detected faults be n_j. If $n_j = 0$, the procedure does not consider s_j any further, and it does not add a stored test to S_i based on s_j in this case. This contributes to a reduction in the number of stored tests. Otherwise, the procedure continues as follows.

For every bit k of s_j, the procedure considers the stored test s_j^k obtained by complementing bit k. For the modified stored test the procedure computes the broadside and skewed-load tests b_j^k and w_j^k. It simulates F under b_j^k and w_j^k, and finds the number of detected faults, n_j^k. If $n_j^k \geq n_j$, the procedure prefers s_j^k over s_j. It assigns $s_j = s_j^k$ and $n_j = n_j^k$. Otherwise, it discards s_j^k.

The procedure considers all the bits of s_j in a random order N_M times, for a constant N_M.

An important difference between the modification of s_j and a random search is that the procedure discards modifications that decrease the number of faults detected by s_j. The search is thus guided by the best stored test obtained to increase the number of detected faults.

The procedure may increase the number of faults detected by a stored test s_j without detecting all the faults that it originally detected. Thus, it is possible that if s_j is considered again, the procedure will again obtain $n_j > 0$. To accommodate this possibility, the procedure considers all the stored tests in S_{i-1} again. This continues as long as the procedure is able to produce additional tests that increase the fault coverage.

In the example of $s298$, the procedure constructs the set S_1 as described in Table II. The first time the procedure considers the stored test $s_0 \in S_0$, the broadside test b_0 detects 31 faults, and the skewed-load test w_0 detects 56 faults, for a total of 87 detected faults. By modifying s_0, the procedure increases the number of faults detected by b_0 to 79. The number of faults detected by w_0 is decreased to 49. This is accepted since the total number of detected faults is increased for s_0 to 128.

TABLE II
IMPROVING EXISTING STORED TESTS

	original		modified	
j	b_j	w_j	b_j	w_j
0	31	56	79	49
1	14	12	36	53
2	6	1	29	23
3	20	14	11	39
4	9	7	14	17
5	1	1	3	29
6	3	1	9	7
7	1	11	1	11
8	3	0	1	8
9	6	0	1	17
10	0	3	5	1
11	6	4	11	5
0	0	7	0	7
1	1	0	2	0
3	5	2	6	2
6	1	0	1	0
10	0	3	2	2
1	1	0	1	0
10	0	3	0	3

The first time s_3 is considered, the procedure reduces the number of faults that b_3 detects from 20 to 11, and increases the number of faults that w_3 detects from 14 to 39. This is accepted since the total number of detected faults is increased for s_3 from 34 to 50.

For the stored tests s_8 and s_9, the skewed-load tests w_8 and w_9 initially do not detect any faults. The procedure increases their numbers of detected faults to 8 and 17, respectively, thus increasing the sharing of input test data between broadside and skewed-load tests.

Similar improvements are obtained for the first 12 tests from S_0. The last test from S_0 does not detect any additional faults, and it does not contribute a stored test to S_1.

The second time the procedure considers the stored test $s_0 \in S_0$, the broadside test b_0 does not detect any faults, and the skewed-load test w_0 detects seven faults. The procedure does not increase the numbers of detected faults.

As the procedure performs additional iterations over the stored tests in S_0, fewer stored tests contribute to S_1. The procedure includes 19 tests in S_1 to achieve an improved fault coverage of 81.38%.

An increase in the number of stored tests occurs when the procedure increases the fault coverage significantly, as in the case of $s298$. When the fault coverage is not increased, or when the increase is small, the procedure reduces the number of stored tests.

D. Using New Random Stored Tests

To improve the fault coverage further after considering all the stored tests in S_{i-1}, the procedure also modifies new random stored tests ($mod2$ in Figure 1). As for the initial test set, the procedure constructs a random stored test s_j by specifying all its bits randomly. It then improves the stored test by modifying its bits one by one. A modified stored test that increases the fault coverage is added to S_i.

The procedure generates $N_{R,1}$ new random stored tests for every number of $LFSR$ bits $l_k \in L$, where $N_{R,1}$ is a parameter of the procedure.

TABLE III
IMPROVING NEW RANDOM STORED TESTS

	original		modified	
j	b_j	w_j	b_j	w_j
1	0	0	1	0
2	0	0	2	0
0	0	0	3	0
1	0	1	0	6
2	0	4	0	4
3	2	0	2	0
4	0	1	0	1
5	0	0	0	4
7	0	0	0	4
8	0	0	0	3

In the example of $s298$, using $N_{R,1} = 10$, the procedure finds two new stored tests for the 4-bit $LFSR$, and eight new stored tests for the 5-bit $LFSR$ as shown in Table III. The value of j is the index of a new stored test. An index is assigned separately for every number of $LFSR$ bits. The fault coverage for $s298$ is increased to 86.41% by adding the stored tests from Table III to S_1. Forward-looking reverse order fault simulation reduces the number of stored tests in S_1 to 26.

It is interesting to note that, for the new stored tests in Table III, either the broadside or skewed-load test detects new faults. Thus, sharing of input test data does not occur. The procedure will improve the sharing of input test data in the step described next, and when it performs another iteration.

E. Eliminating Stored Tests that Detect Single Faults

In the example of $s298$, after constructing S_1, the set of stored tests contains two tests that detect single faults, s_{15} that detects f_{305}, and s_{17} that detects f_{221}. Sharing of input test data does not occur for such tests, and they increase the number of tests while detecting only one fault. The procedure attempts to eliminate such tests from the test set in order to reduce the number of tests and increase the sharing of input test data ($mod3$ in Figure 1).

It should be noted that numbers of detected faults are determined by fault simulation with fault dropping. The step of removing tests that detect single faults is described next.

The procedure considers every test $s_j \in S_i$ that detects only one fault. Let the single fault detected by s_j be f_j. The procedure considers every other test $s_m \in S_i$ that has not been marked for removal from the test set. Using the same modification process as above, the procedure attempts to modify s_m such that it would continue to detect the same faults, and in addition, detect f_j. When this is successful, the procedure marks that f_j is detected by s_m, and that s_j can be removed from the test set.

Stored tests at the beginning of S_i detect relatively large numbers of faults. Such tests are difficult to modify so as to detect an additional fault. To avoid unnecessary computations, the procedure considers for modification a stored test s_m only if $m \geq N_S$, for a parameter N_S.

This step changes the stored tests in S_i. Therefore, the procedure performs fault simulation of the modified set even if no test is removed in order to capture the increase in the fault coverage.

TABLE IV
COMPARISON OF TEST SETS

circuit	tests	f.c.
s1423	50	98.66
s5378	170	97.86
s9234	258	93.32
s13207	373	96.83
s15850	300	95.13
s35932	34	89.78
b04	42	99.78
b14	267	94.55
b20	355	95.77
s38584	393	93.79
b15	423	97.32
b21	468	92.36
b22	528	94.97
aes_core	273	99.99
des_area	112	100.00
i2c	84	98.81
sasc	34	99.77
simple_spi	56	99.14
spi	460	99.70
systemcaes	158	99.84
systemcdes	74	99.98
tv80	663	97.86
usb_phy	40	99.05
wb_dma	138	99.68

In the example of $s298$, without limiting the tests that will be considered for modification, the procedure modifies s_{18} to detect f_{305}. It thus removes s_{15} from S_1.

IV. EXPERIMENTAL RESULTS

The procedure illustrated by Figure 1 was applied to transition faults in benchmark circuits using the following parameter values.

For a circuit with K state variables, the set L of $LFSR$s consists of primitive $LFSR$s from [23] with numbers of bits $b_0 = 4$, $b_1 = 5$, ..., $b_{N_L-1} = min\{K/4, 130\}$. The upper bound of $K/4$ ensures that a stored seed requires at most a fourth of the number of bits required for storing a scan-in state uncompressed. For larger circuits the upper bound of 130 ensures a smaller ratio.

In the worst case, a 130-bit programmable $LFSR$ requires 129 XOR gates and the same number of multiplexers. Seven bits are required for encoding 127 $LFSR$s with four to 130 bits. Thus, the encoding of an $LFSR$ requires a seven-input AND gate. The AND gate is also driven by the last bit of the $LFSR$. This yields a total of at most 127 eight-input AND gates as part of the multiplexers. Assuming that an $LFSR$ has three non-zero terms, which is the maximum in [23], all the multiplexers together have a total of $3 \cdot 127 = 381$ OR gate inputs. This worst case does not occur for any of the benchmark circuits considered.

The procedure considers $N_{R,0} = 1000$ random stored tests for the initial set S_0, and $N_{R,1} = 5$ random stored tests for improving a set S_i when $i > 0$.

The procedure improves a stored test by considering its bits $N_M = 4$ times. It considers tests for modification starting with $N_S = |S_i|/4$. It terminates after $N_I = 4$ iterations that do not improve the set of stored tests. These parameter values were selected based on experiments showing that larger parameter

TABLE V
EXPERIMENTAL RESULTS (WITH AND WITHOUT SHARING)

circuit	sv	iter	rand	rem	tests	share	incr	$LFSR$s	bits	storage	decr	f.c.	incr	ntime
sasc	117	0	0	0	80	67.50	0.00	8	5.74	2859	0.00	99.772	0.000	1.00
sasc	117	1	0	0	72	62.50	-5.00	8	6.18	2605	8.88	99.772	0.000	5.96
sasc	117	4	0	0	59	66.10	-1.40	8	6.36	2145	24.97	99.772	0.000	17.87
sasc	117	10	0	0	73	0.00	0.00	6	5.89	2620	23.21	99.772	0.000	35.31
des_area	128	0	0	0	279	56.27	0.00	2	4.01	134482	0.00	100.000	0.000	1.00
des_area	128	1	0	5	115	53.91	-2.36	1	4.00	55430	58.78	100.000	0.000	339.29
des_area	128	6	0	4	104	61.54	5.27	1	4.00	50128	62.73	100.000	0.000	1766.46
des_area	128	4	0	2	114	0.00	0.00	1	4.00	54948	63.81	100.000	0.000	1128.20
usb_phy	98	0	0	0	82	68.29	0.00	9	6.55	2833	0.00	99.049	0.000	1.00
usb_phy	98	1	0	0	66	74.24	5.95	9	7.15	2320	18.11	99.049	0.000	4.20
usb_phy	98	5	0	0	82	0.00	0.00	9	7.56	2916	11.80	99.049	0.041	16.63
systemcdes	190	0	0	0	144	73.61	0.00	3	4.27	38055	0.00	99.983	0.000	1.00
systemcdes	190	1	0	1	78	83.33	9.72	3	4.14	20603	45.86	99.983	0.000	47.52
systemcdes	190	2	0	1	69	86.96	13.35	3	4.16	18227	52.10	99.983	0.000	93.28
systemcdes	190	9	0	3	111	0.00	0.00	3	4.14	29319	45.34	99.983	0.000	417.10
s1423	74	0	0	0	102	56.86	0.00	13	6.79	4161	0.00	98.278	0.000	1.00
s1423	74	1	0	2	82	60.98	4.11	12	6.54	3324	20.12	98.349	0.070	6.97
s1423	74	9	0	0	68	72.06	15.20	12	7.56	2826	32.08	98.665	0.387	41.57
s1423	74	20	0	0	91	0.00	0.00	13	7.81	3805	19.35	98.665	0.281	99.57
s13207	669	0	0	0	698	34.24	0.00	103	29.25	63692	0.00	94.370	0.000	1.00
s13207	669	1	15	14	620	39.52	5.28	97	33.12	58973	7.41	95.777	1.408	16.10
s13207	669	30	0	12	439	52.16	17.92	41	54.74	51247	19.54	96.832	2.462	348.16
s13207	669	25	0	9	552	0.00	0.00	70	52.38	63140	10.88	96.832	2.060	243.90
b04	66	0	0	0	67	50.75	0.00	8	5.27	1961	0.00	99.650	0.000	1.00
b04	66	1	0	0	48	64.58	13.84	7	5.23	1403	28.45	99.781	0.131	4.17
b04	66	2	0	0	45	73.33	22.59	7	5.22	1315	32.94	99.825	0.175	7.33
b04	66	7	0	0	60	0.00	0.00	8	5.25	1755	25.73	99.825	0.482	22.58

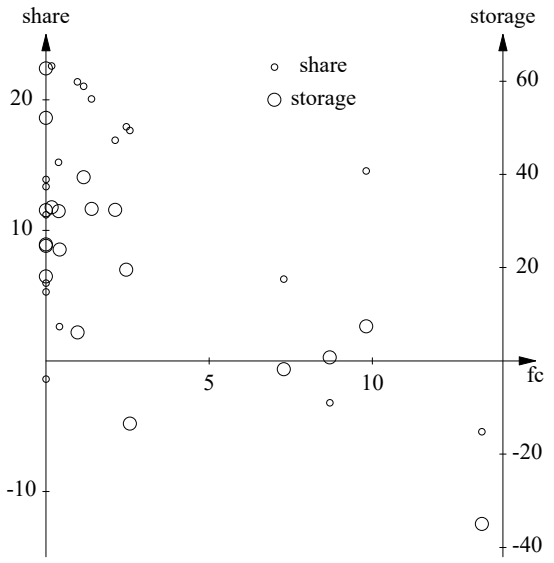

Fig. 2. Percentage of sharing and storage

values increase the computational effort but do not contribute significantly to the quality of the results.

As a reference point, Table IV provides the number of tests and the transition fault coverage for one of two test sets that consists of broadside and skewed-load tests. (1) The test set from [12] when it is available. (2) A test set computed without requiring broadside and skewed-load tests to share input test data. In both cases, the constraints of a test data compression method are not considered. It is important to note that the constraints of a test data compression method are known to increase the number of stored tests. They can also reduce the achievable fault coverage.

For further comparison, the procedure from Figure 1 is applied without sharing of input test data between broadside and skewed-load tests. Every stored test s_j is associated with a single test type, and only a test of this type is applied based on s_j. The same property is maintained as stored tests are modified to increase the number of faults they detect. The number of random stored tests considered is doubled to allow random stored tests to achieve a similar fault coverage.

The results of the procedure from Figure 1 are given in Tables V and VI. On up to three separate rows, the results are reported for S_0, S_1, and the iteration where the final set of stored tests is obtained with sharing of input test data. The results for additional iterations are given in Table VI for two circuits, spi and wb_dma, to demonstrate the possibility of terminating the procedure earlier. For several circuits in Table V, an additional row shows the final results obtained without sharing of input test data.

For every set, after the circuit name, column sv shows the number of state variables. Column $iter$ shows the iteration i of the procedure. For $i > 0$, column $rand$ shows the number of new modified random stored tests that are added to S_i. Column rem shows the number of tests that detect single faults and the procedure removes from the set.

Column $tests$ shows the number of stored tests in S_i. Column $share$ shows the percentage of stored tests for which both the broadside and skewed-load test increases the fault coverage, and needs to be applied. A higher percentage implies better sharing of input test data between broadside and skewed-load tests. Column $incr$ that follows shows the increase in the

TABLE VI
EXPERIMENTAL RESULTS (WITH SHARING)

circuit	sv	iter	rand	rem	tests	share	incr	$LFSR$s	bits	storage	decr	f.c.	incr	ntime
b14	247	0	0	0	334	47.60	0.00	43	13.19	26448	0.00	81.583	0.000	1.00
b14	247	1	7	6	261	56.70	9.10	38	13.64	20787	21.40	84.674	3.091	4.82
b14	247	56	0	2	384	42.19	-5.42	26	26.96	35698	-34.97	94.936	13.353	291.27
b22	709	0	0	0	827	51.03	0.00	92	20.56	71582	0.00	86.377	0.000	1.00
b22	709	1	24	16	635	63.62	12.59	80	25.10	57850	19.18	89.093	2.716	5.71
b22	709	7	2	16	870	46.55	-4.48	71	51.11	101884	-42.33	94.575	8.198	36.17
i2c	128	0	0	0	132	59.09	0.00	25	9.61	5756	0.00	90.163	0.000	1.00
i2c	128	1	5	0	114	55.26	-3.83	25	12.25	5272	8.41	94.406	4.242	3.23
i2c	128	33	0	1	102	55.88	-3.21	16	21.99	5711	0.78	98.858	8.695	80.92
b21	494	0	0	0	568	46.13	0.00	86	22.46	50247	0.00	81.104	0.000	1.00
b21	494	1	22	13	472	55.30	9.17	83	27.17	43974	12.48	84.397	3.292	5.25
b21	494	36	0	13	645	43.57	-2.56	37	72.98	89641	-78.40	93.098	11.993	195.53
b20	494	0	0	0	565	46.73	0.00	78	19.54	48330	0.00	86.237	0.000	1.00
b20	494	1	13	8	447	62.42	15.69	74	22.82	39702	17.85	88.907	2.671	4.60
b20	494	19	1	6	582	44.85	-1.88	46	53.25	69401	-43.60	95.056	8.820	73.25
s5378	179	0	0	0	412	39.08	0.00	38	12.37	33935	0.00	97.337	0.000	1.00
s5378	179	1	3	4	357	36.97	-2.10	40	13.10	29668	12.57	97.517	0.179	35.49
s5378	179	10	0	5	295	41.69	2.62	33	17.59	25838	23.86	97.753	0.415	251.07
s35932	1728	0	0	0	63	92.06	0.00	3	4.37	4685	0.00	89.781	0.000	1.00
s35932	1728	1	0	0	55	90.91	-1.15	3	4.38	4091	12.68	89.781	0.000	4.00
s35932	1728	7	0	0	49	95.92	3.85	3	4.43	3647	22.16	89.781	0.000	21.81
s9234	228	0	0	0	491	45.62	0.00	53	18.57	27776	0.00	85.851	0.000	1.00
s9234	228	1	19	8	464	45.91	0.28	52	21.95	27816	-0.14	89.625	3.774	6.10
s9234	228	40	0	8	399	51.88	6.26	31	32.86	28274	-1.79	93.134	7.283	176.83
aes_core	530	0	0	0	666	78.83	0.00	8	7.26	348489	0.00	99.992	0.000	1.00
aes_core	530	1	0	1	579	82.73	3.90	8	7.58	303153	13.01	99.992	0.000	220.28
aes_core	530	10	0	0	501	90.02	11.19	8	8.07	262561	24.66	99.992	0.000	1467.82
b15	447	0	0	0	782	50.26	0.00	104	28.38	78494	0.00	88.939	0.000	1.00
b15	447	1	37	12	743	48.72	-1.53	101	39.58	82903	-5.62	96.901	7.962	7.24
b15	447	51	0	10	554	64.80	14.55	76	59.19	72681	7.41	98.749	9.810	280.97
systemcaes	670	0	0	0	370	59.19	0.00	21	8.06	193903	0.00	99.841	0.000	1.00
systemcaes	670	1	0	3	309	66.02	6.83	21	8.70	162131	16.39	99.841	0.000	102.13
systemcaes	670	14	0	2	249	73.09	13.90	19	11.26	131287	32.29	99.841	0.000	1125.24
tv80	359	0	0	0	1191	22.17	0.00	86	28.85	65324	0.00	92.528	0.000	1.00
tv80	359	1	47	18	1076	27.42	5.25	84	32.31	62739	3.96	96.219	3.691	14.77
tv80	359	47	0	21	836	36.36	14.20	53	46.70	60775	6.96	98.423	5.895	584.52
simple_spi	131	0	0	0	130	58.46	0.00	15	6.34	4724	0.00	97.068	0.000	1.00
simple_spi	131	1	3	1	100	58.00	-0.46	15	6.96	3696	21.76	97.304	0.236	5.27
simple_spi	131	46	0	0	69	75.36	16.90	16	16.29	3194	32.39	99.188	2.120	140.10
s15850	597	0	0	0	524	42.18	0.00	95	26.30	28455	0.00	92.475	0.000	1.00
s15850	597	1	29	13	481	42.00	-0.18	95	35.44	30516	-7.24	93.791	1.316	5.63
s15850	597	45	0	9	351	59.83	17.65	30	63.97	32281	-13.45	95.043	2.568	164.71
s38584	1452	0	0	0	1188	57.91	0.00	108	24.19	57253	0.00	92.804	0.000	1.00
s38584	1452	1	24	27	979	66.39	8.48	102	28.02	50929	11.05	93.296	0.492	7.54
s38584	1452	20	0	17	705	79.29	21.38	76	52.23	53740	6.14	93.768	0.964	103.10
spi	229	0	0	0	740	49.59	0.00	37	9.77	73832	0.00	98.329	0.000	1.00
spi	229	1	3	11	621	53.30	3.71	33	11.15	62816	14.92	99.332	1.003	68.81
spi	229	2	3	7	577	56.67	7.08	33	12.28	59016	20.07	99.515	1.186	123.35
spi	229	5	2	7	508	65.94	16.35	33	14.64	53158	28.00	99.641	1.312	244.60
spi	229	12	0	6	477	67.30	17.70	31	16.28	50694	31.34	99.699	1.370	501.37
spi	229	24	0	9	468	69.66	20.06	27	16.38	49788	32.57	99.724	1.395	945.93
wb_dma	523	0	0	0	353	46.46	0.00	69	22.03	159567	0.00	98.605	0.000	1.00
wb_dma	523	1	5	12	297	52.19	5.73	61	28.76	136253	14.61	99.402	0.797	48.65
wb_dma	523	2	2	6	271	54.98	8.52	56	30.59	124820	21.78	99.456	0.852	81.64
wb_dma	523	5	0	6	245	59.18	12.72	53	33.84	113641	28.78	99.607	1.003	158.58
wb_dma	523	13	1	6	228	59.65	13.19	49	40.18	107201	32.82	99.746	1.142	322.61
wb_dma	523	23	0	6	206	67.48	21.02	43	39.50	96718	39.39	99.758	1.154	527.03

percentage of sharing. The circuits are arranged by ascending order of this parameter.

Column $LFSRs$ shows the number of $LFSR$s from L that are used by the stored tests in S_i. Column $bits$ shows the average number of bits in a seed of the stored test set. Denoting the average by l_{ave} and the number of state variables by K, the level of test data compression obtained per test is equal to K/l_{ave}. The selection of L ensures that this value is at least four for smaller circuits, and larger than four for larger circuits.

Column $storage$ shows the number of bits required for storing S_i. The number of bits includes the seeds and the primary input vectors. Assuming that every stored test is used for applying both a broadside and a skewed-load test, and the two test types are applied separately, no extra bits are included for the test type (two bits are required for every stored test to indicate which one of the two test types should be applied, and this is negligible compared with the seeds and primary

TABLE VII
EXPERIMENTAL RESULTS (SMALLER SETS OF $LFSR$s)

circuit	sv	iter	rand	rem	tests	share	incr	$LFSR$s	bits	storage	decr	f.c.	incr	ntime
s5378	179	0	0	0	412	39.08	0.00	38	12.37	33935	0.00	97.337	0.000	1.00
s5378	179	10	0	5	295	41.69	2.62	33	17.59	25838	23.86	97.753	0.415	251.07
s5378	179	0	0	0	365	43.01	0.00	8	38.71	39680	0.00	95.713	0.000	1.00
s5378	179	20	0	2	241	46.89	3.87	6	37.43	25891	34.75	97.875	2.162	1457.84
s5378	179	0	0	0	354	42.37	0.00	8	16.87	30752	0.00	95.713	0.000	1.00
s5378	179	39	0	0	263	39.92	-2.45	8	16.79	22827	25.77	97.781	2.068	3285.47

input vectors). Column $decr$ shows the percentage reduction in the number of storage bits required for S_i relative to S_0.

Column $f.c.$ shows the transition fault coverage achieved by S_i. Column $incr$ shows the increase in the transition fault coverage relative to S_0. A larger increase in the fault coverage sometimes requires more stored tests, and a higher number of bits for storage of S_i. It can also reduce the sharing of input test data between broadside and skewed-load tests.

Column $ntime$ shows the normalized runtime, where the cumulative runtime for computing S_0, S_1, ..., S_i is divided by the runtime for computing S_0. The computation of S_0 is dominated by fault simulation with fault dropping of $2N_L N_{R,0}$ tests.

The following points can be seen from Tables V and VI. The procedure increases the fault coverage of S_0 when it is lower than the highest fault coverage achievable by broadside and skewed-load tests. The increase is significant for several of the circuits. Thus, the procedure is able to provide a meaningful fault coverage increase when needed. In most of the cases, the final fault coverage is equal or close to that achievable by broadside and skewed-load tests that are not constrained by a test data compression method.

The parameter that measures the ability of the procedure to share input test data between broadside and skewed-load tests is the percentage of sharing shown under column $share$. This is the percentage of stored tests for which both the broadside and skewed-load test needs to be applied. An increase in the fault coverage, or a reduction in the number of storage bits, can hide an improvement in the percentage of sharing. Nevertheless, this parameter is improved for many of the circuits in Tables V and VI.

An increase in the fault coverage can also hide a reduction in the number of storage bits because more stored tests are needed to support the increased fault coverage. This results in a negative number under column $decr$. Nevertheless, the reduction in the number of storage bits is significant in many cases. Moreover, when the procedure is not allowed to share input test data between broadside and skewed-load tests, the number of stored tests, and the storage requirements of the stored test set, are significantly higher.

For further illustration of the procedure with sharing of input test data, Figure 2 shows the percentage increase in sharing and the percentage reduction in storage as a function of the increase in the fault coverage considering several of the circuits from Tables V and VI. The increase in the fault coverage is shown on the horizontal axis. On the vertical axis, the small circles show the increase in the percentage of sharing, and the large circles show the percentage reduction in storage. Figure 2 shows that a reduction in the percentage of sharing, and an increase in the number of storage bits, typically go together with a high increase in the fault coverage.

The average number of $LFSR$ bits typically increases as the procedure performs additional iterations. The number of storage bits decreases for many of the circuits because the number of stored tests is reduced significantly when somewhat larger $LFSR$s are used. The number of different $LFSR$s typically decreases as the average number of $LFSR$ bits is increased.

The normalized runtime after one iteration is similar for circuits of different sizes. This implies that the procedure scales similar to a fault simulation procedure. It should be noted in this regard that the procedure modifies seeds, and not scan-in states that have a significantly larger number of bits. The average number of bits in a seed (the average number of $LFSR$ bits) is similar for circuits of different sizes.

The normalized runtime per iteration typically decreases as the procedure performs additional iterations. This occurs because the number of tests is reduced, and in spite of the increase in the average number of $LFSR$ bits.

The procedure can be terminated earlier without a significant impact on its effectiveness. This is illustrated by spi and wb_dma in Table VI. The earlier termination can be implemented using a runtime limit or a lower value of N_I.

Finally, it is possible to reduce the set of $LFSR$s that the procedure is allowed to use in order to reduce the hardware overhead associated with the programmable $LFSR$. To demonstrate this point, the procedure is run for $s5378$ with two sets of eight $LFSR$s. The first set is defined by $b_0 = 37$, $b_1 = 38$, ..., $b_7 = 44$. For the second set, $b_0 = 15$, $b_1 = 16$, ..., $b_7 = 22$. These sets were selected based on the maximum and average number of $LFSR$ bits obtained for $s5378$ in Table VI.

Table VII shows the results obtained for $s5378$ with these sets of $LFSR$s. The results from Table VI are repeated for ease of comparison. The results in every case are shown for S_0 and for the final iteration of the procedure. It can be seen from Table VII that the procedure is effective with a limited set of $LFSR$s as well.

V. CONCLUDING REMARKS

This paper developed an algorithm for computing stored tests that can be used for applying both broadside and skewed-load tests. The algorithm was developed for the case where a programmable linear-feedback shift-register ($LFSR$) is used for on-chip decompression. A stored test consists of a seed for the $LFSR$ and two primary input vectors. The seed determines

the scan-in state of the applied broadside and skewed-load tests, as well as the additional scan-in vector required for the skewed-load test. In the algorithm developed in this paper, stored tests are computed directly without first computing broadside or skewed-load tests. This avoids situations where the tests cannot be compressed or do not have common input test data. The algorithm consisted of the generation of random stored tests, and the modification of stored tests to increase the fault coverage contribution achieved by the broadside and skewed-load tests they apply. Experimental results for transition faults in benchmark circuits demonstrated the ability of the procedure to share stored tests between broadside and skewed-load tests while increasing the fault coverage and reducing the number of stored tests and their storage requirements.

REFERENCES

[1] B. Koenemann, "LFSR-Coded Test Patterns for Scan Designs", in Proc. Europ. Test Conf., 1991, pp. 237-242.

[2] S. Hellebrand, S. Tarnick, J. Rajski and B. Courtois, "Generation of Vector Patterns Through Reseeding of Multiple-Polynomial Linear Feedback Shift Register", in Proc. Intl. Test Conf., 1992, pp. 120-129.

[3] K.-J. Lee, J. Chen and C. Huang, "Using a Single Input to Support Multiple Scan Chains", in Proc. Intl. Conf. Computer-Aided Design, 1998, pp. 74-78.

[4] C. Barnhart, V. Brunkhorst, F. Distler, O. Farnsworth, B. Keller and B. Koenemann, "OPMISR: The Foundation for Compressed ATPG Vectors", in Proc. Intl. Test Conf., 2001, pp. 748-757.

[5] J. Rajski, J. Tyszer, M. Kassab, N. Mukherjee, R. Thompson, K.-H. Tsai, A. Hertwig, N. Tamarapalli, G. Mrugalski, G. Eide and J. Qian, "Embedded Deterministic Test for Low Cost Manufacturing Test", in Proc. Intl. Test Conf., 2002, pp. 301-310.

[6] N. A. Touba, "Survey of Test Vector Compression Techniques", IEEE Design & Test, Apr. 2006, pp. 294-303.

[7] V. Tenentes, X. Kavousianos and E. Kalligeros, "State Skip LFSRs: Bridging the Gap between Test Data Compression and Test Set Embedding for IP Cores", in Proc. Design, Automation and Test in Europe Conf., 2008, pp. 474-479.

[8] S. Alampally, R. T. Venkatesh, P. Shanmugasundaram, R. A. Parekhji and V. D. Agrawal, "An Efficient Test Data Reduction Technique through Dynamic Pattern Mixing Across Multiple Fault Models", in Proc. VLSI Test Symp., 2011, pp. 285-290.

[9] I. Pomeranz and S. M. Reddy, "Static Test Data Volume Reduction Using Complementation or Modulo-M Addition", IEEE Trans. on VLSI Systems, June 2011, pp. 1108-1112.

[10] D. Czysz, G. Mrugalski, N. Mukherjee, J. Rajski, P. Szczerbicki and J. Tyszer, "Deterministic Clustering of Incompatible Test Cubes for Higher Power-Aware EDT Compression", IEEE Trans. on Computer-Aided Design, Aug. 2011, pp. 1225-1238.

[11] A. Chandra, J. Saikia and R. Kapur, "Breaking the Test Application Time Barriers in Compression: Adaptive Scan-Cyclical (AS-C)", in Proc. Asian Test Symp, 2011, pp. 432-437.

[12] I. Pomeranz, "On the Computation of Common Test Data for Broadside and Skewed-Load Tests", IEEE Trans. on Computers, April 2012, pp. 578-583.

[13] O. Acevedo and D. Kagaris, "Using the Berlekamp-Massey Algorithm to Obtain LFSR Characteristic Polynomials for TPG", in Proc. Intl. Symp. on Defect and Fault Tolerance in VLSI and Nanotechnology Systems, 2012, pp. 233-238.

[14] X. Lin and J. Rajski, "On Utilizing Test Cube Properties to Reduce Test Data Volume Further", in Proc. Asian Test Symp., 2012, pp. 83-88.

[15] I. Pomeranz, "Input Test Data Volume Reduction for Skewed-Load Tests by Additional Shifting of Scan-In States", IEEE Trans. on Computer-Aided Design, April 2014, pp. 638-642.

[16] I. Pomeranz, "On the Use of Multi-Cycle Tests for Storage of Two-Cycle Broadside Tests", in Proc. VLSI Test Symp., 2014.

[17] I. Pomeranz, "Computation of Seeds for $LFSR$-Based Diagnostic Test Generation", IEEE Trans. on Computer-Aided Design, Dec. 2015, pp. 2004-2012.

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

[19] I. Pomeranz, "$LFSR$-Based Test Generation for Path Delay Faults", IEEE Trans. on Computer-Aided Design, Feb. 2019, Vol. 38, No. 2, pp. 345-353.

[20] J. Savir and S. Patil, "Scan-Based Transition Test", IEEE Trans. on Computer-Aided Design, Aug. 1993, pp. 1232-1241.

[21] J. Savir and S. Patil, "Broad-Side Delay Test", IEEE Trans. on Computer-Aided Design, Aug. 1994, pp. 1057-1064.

[22] I. Pomeranz and S. M. Reddy, "On Test Generation with Test Vector Improvement", IEEE Trans. on Computer-Aided Design, March 2010, pp. 502-506.

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

PLACE PHOTO HERE

Irith Pomeranz (M'89-SM'96-F'99) received the B.Sc degree (Summa cum Laude) in Computer Engineering and the D.Sc degree from the Department of Electrical Engineering at the Technion - Israel Institute of Technology in 1985 and 1989, respectively. From 1989 to 1990 she was a Lecturer in the Department of Computer Science at the Technion. From 1990 to 2000 she was a faculty member in the Department of Electrical and Computer Engineering at the University of Iowa. In 2000 she joined the School of Electrical and Computer Engineering at Purdue University. Her research interests include testing of VLSI circuits, design for testability, synthesis and design verification. Dr. Pomeranz is a recipient of the NSF Young Investigator Award in 1993, and of the University of Iowa Faculty Scholar Award in 1997. Three of her conference papers won best paper awards, and four other papers were nominated for best paper awards. One of the papers she co-authored was selected by the 2016 International Test Conference as the most significant paper published ten years before. She delivered a keynote speech at the 2006 Asian Test Symposium. She served as associate editor of the ACM Transactions on Design Automation, the IEEE Transactions on Computers, and the IEEE Transactions on VLSI Systems. She served as guest editor of the IEEE Transactions on Computers January 1998 special issue on Dependability of Computing Systems, and as program co-chair of the 1999 Fault-Tolerant Computing Symposium. She served as program chair of the 2004 and 2005 VLSI Test Symposium, and as general chair of the 2006 VLSI Test Symposium. She is a Golden Core Member of the IEEE Computer Society.