Selective-Run Built-In Self-Test Using an Embedded Processor *

Sungbae Hwang and Jacob A. Abraham
Computer Engineering Research Center
The University of Texas at Austin
Austin, TX 78712
{hsb, jaa}@cerc.utexas.edu

ABSTRACT

Many systems-on-a-chip (SOCs) include processors as central units to implement diverse algorithms and control peripheral units such as embedded cores. The computing power of the embedded processor can be used to self-test its own functions as well as to test the other cores within the chip boundary. In BIST methodology, pseudo-random pattern testing can reduce the memory requirements. In addition to general pseudo-random pattern testing, this paper proposes and evaluates a novel selective-random pattern test technique. This technique increases the fault coverage while significantly reducing test application time. This also greatly decreases the memory requirements compared to traditional BIST schemes. The cost for extra hardware is low and the technique is easily integrated with parallel scan and boundary scan designs.

Categories and Subject Descriptors

B.7.3 [**Integrated Circuits**]: Reliability and Testing—*Built-in tests, Test generation*

Keywords

SOC testing, built-in self-test, design for testability, processor-based testing, pseudo-random number generator.

1. INTRODUCTION

Advances in VLSI technology has made possible systems on a single chip, and core-based SOC designs have provided a solution for reducing product cycles and system costs and hence for meeting consumer demand. Cores are predesigned reusable building blocks, and system designers can readily integrate those cores to compose a meaningful system. However, at the same time, this approach poses other problems [1]. As cores become more complex and the number of cores that are integrated on a chip increases, the volume of test data increases rapidly. The I/O channel capacity significantly limits the accessibility to the cores from the external test equipment.

Many researchers have been studying how to alleviate those problems. While Ishida [2] and Yamaguchi [3] proposed data compression techniques in which test data were decompressed by external test equipment, Chandra [4] and Jas [5] suggested on-chip DFT circuits to decompress test data. Many other researchers introduced BIST circuits in which test data were automatically generated mostly by the pseudo-random pattern generator logic [6, 7]. Others tried to reuse on-chip processors or data-path circuits which were included as function blocks of the chip [8, 9, 10] Among these on-chip processor approaches can be considered full-fledged DFT and BIST techniques. Once the processor is verified to work correctly, its use for testing other cores gives significant advantages over the other techniques. Its programmability allows a software driven BIST technique. Its data-path enables many arithmetic and logical operations. Many test pattern generation methods, which are generally realized in hardware circuits such as linear feedback shift registers (LFSRs) and cellular automata (CAs), or other even more complex generation algorithms, can be employed in a software form using an on-chip processor. Hence this approach can reduce the silicon area overhead.

In [10], a technique was introduced in which the embedded processor can access I/O terminals of other embedded cores in test mode via an already existing system bus, thus reducing the silicon area overhead. This technique can incorporate BIST schemes through software, but introduces large memory requirements to store the test data. [11] proposed a BIST technique that could apply test patterns onto other cores in parallel and utilize pseudo-random number generators (PRNGs). The technique reduces the memory requirements on test data; by using a parallel access mechanism it decreases the testing time. In this paper, we propose a more advanced technique that significantly reduces the testing time as well as achieves the targeted fault coverage with a small number of test applications, while using a very simple pseudo-random number generator. The required memory is so small that it can be easily implemented on internal memory such as cache. It also utilizes the existing system bus as the test access mechanism; hence the area overhead is greatly reduced.

For testing of DSP cores, Radecka et al. proposed an

*This research was supported in part by the Semiconductor Research Corporation under contract 99-TJ-708.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GLSVLSI'02, April 18-19, 2002, New York, New York, USA.
Copyright 2002 ACM 1-58113-462-2/02/0004 ...$5.00.

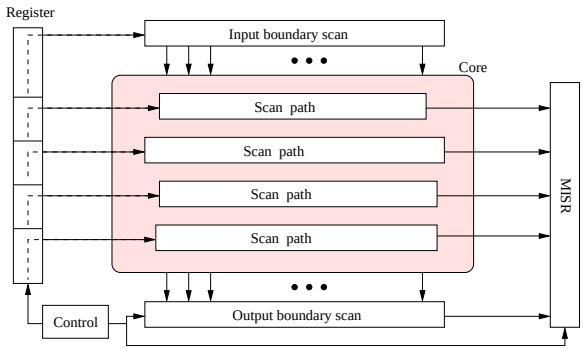

Figure 1: BIST architecture for each core.

arithmetic built-in self-test (ABIST) scheme in [12]. Their methodology uses the programmability of DSP cores to test peripheral devices. However, they assumed that a dedicated test access mechanism exists from the DSP core to each core under test, which we do not need. Our approach is also different from [8, 12] in that we use a test pattern selection mechanism to reduce the test application time for the random testing. Compared to their combined method of BIST and deterministic technique, our technique requires far less memory usage. Papachristou et al. suggested microprocessor-based testing for core-based SOCs in [13]. They focused on how to achieve test paths from the processor, bypassing the other cores in between, and used graph modeling to get the shortest path. Our approach is simpler, since we reuse the system bus in test mode as well as in functional mode.

The paper is organized as follows. Section 2 describes the BIST architecture based on the embedded processor, and Section 3 shows the implementation of several pseudo-random number generators. Section 4 proposes the new methodology that selectively applies test patterns and hence reduces the testing time in random pattern testing. Experimental results are in Section 5. Conclusions are given in Section 6

2. BIST ARCHITECTURE

Testing the processor and the test memory is not the focus of our work. We assume that the processor and the test memory are tested by some other methods [14, 15, 16].

The system bus can function as the test access mechanism in test mode [10]. The I/O terminals of the cores are accessible from the processor in the same way as the internal registers of the cores. The processor uses the same protocol to read and write those I/O terminal registers of each core.

Hwang [11] presented a BIST methodology using the system bus and showed that it only requires small area overhead. It incorporated a register for each wrapper, which also could be accessed from the embedded processor. In this paper, we also use the same registers to enhance fault coverage in BIST mode. We will briefly introduce the architecture.

In our technique, once a value is loaded onto a core's register, it is then shifted into the core through both the boundary scan and the internal scan. Figure 1 shows the BIST architecture in which the register is loaded from the processor and the value in the register is shifted into the attached core. The register is divided into several fields each of which is taking care of a scan path. A scan path gets its input value from the corresponding field of the register.

Table 1: Control register

Name	Description
TM	This *test mode* allows to select between test mode(1) and normal mode(0)
BM	When in test mode (TM=1), this allows to select between BIST mode(1) and direct access mode(0)
RUN	If it is written with 1, then the core run one clock in functional mode. It is automatically cleared.
STOP	This bit blocks the scan shifting. It is cleared when RUN=1.

Each dashed line illustrates the shifting flow of bit values in a field. The size of each field does not need to be the same if the core has a separate enable signal for each scan path. Though it is not shown in Figure 1, the I/O registers are also assumed to be directly accessible from the processor. After loading a test vector using the register, the processor signals the core to run in functional mode and to capture the response on the internal scan cells and output terminal registers. Then the next vector is loaded again from the processor while the response from the previous vector is compressed on the multiple input signature register (MISR).

In order to achieve the controllability mentioned earlier, each test wrapper contains a control register. Table 1 shows several fields in the control register [11].

3. PSEUDO-RANDOM NUMBER GENERATOR

This section discusses several pseudo-random number generators for BIST implementation. We also show efficient implementations on an embedded processor.

BIST schemes using processors have different features from those using hardware BIST. In hardware BIST, by far the most popular pseudo-random pattern generator is LFSRs. Due to structural dependencies, the use of LFSRs as two-dimensional test pattern generators feeding a multiplicity of scan chains in parallel may result in unsatisfactory fault coverage [17]. If the scan paths are fed directly from adjacent bits of the LFSR, then this very close proximity will cause neighboring scan chains to contain test patterns which are highly correlated. In order to alleviate this problem, phase shifters are used between LFSRs and the circuit under test. Alternatively, CAs can be used for the pseudo-random patterns. A CA evolves in discrete steps with the next value of one site determined by its previous value and that of a set of sites called the neighbor sites. Wolfram [18] did the basic classification of CA based on their qualitative and functional behavior.

Some linear hybrid CAs (LHCAs) can provide the maximal length binary sequence [19]. One of those LHCAs is composed of rules 90 and 150 which are defined as follows:

$$\text{Rule 90}: \quad s_i^+ = s_{i-1} \oplus s_{i+1} \tag{1}$$

$$\text{Rule 150}: \quad s_i^+ = s_{i-1} \oplus s_i \oplus s_{i+1} \tag{2}$$

where s_i^+ denotes the next state for site s_i. The local rules for a one-dimensional neighborhood-three cellular automaton are described by an eight-digit binary number, as in the example of Figure 2. The eight possible states of three ad-

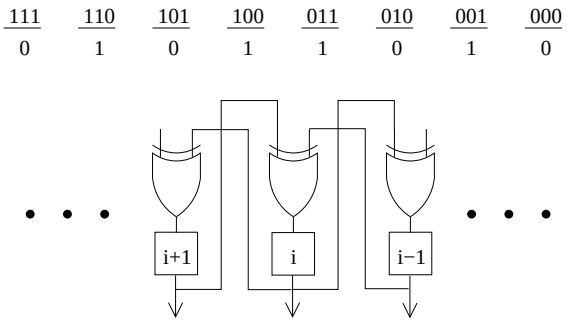

Figure 2: A rule 90 CA.

jacent sites are given above the horizontal lines. The one digit beneath each line represents the next value of the central site. The one-digit values compose the rule number together. For example, Figure 2 shows "01011010", which is *90* in decimal number, and its hardware implementation.

An LHCA is a CA in which some sites follow a rule while the others follow another rule. LHCAs can be easily implemented in software as well as in hardware. The following code shows the implementation on *ARM* processor assuming that `R0` contains the current value and `R2` contains the encoded LHCA value for the rules. The shift operations (`LSL` and `LSR` mean logical shift left and logical shift right, respectively) are used to get the neighbors' values and the AND operation is for the selection between CA rules 90 and 150.

```
MOV   R1, R0, LSL #1     ; R1=R0<<1
XOR   R1, R1, R0, LSR #1; R1^=(R0>>1)
AND   R0, R0, R2         ; R0&=R2
XOR   R0, R0, R1         ; R0^=R1
```

Radecka et al. suggested a random number generator in [12] in the context of their ABIST scheme as follows.

$$A_i \;=\; A^l_{i-1} \cdot \mathcal{M} \;+\; A^h_{i-1} \qquad (3)$$

where A^l_i and A^h_i are the contents of the n least significant and n most significant bits of A, respectively, after i iterations. $\mathcal{M}$ is an n-bit constant. This generator was originally developed to utilize datapath blocks in processors. Its ARM implementation is quite easy as follows.

```
AND   R3, R1, R5       ; Get 16 LSBs
MOV   R4, R1, LSR #16 ; Get 16 MSBs
MLA   R1, R2, R3, R4  ; R1=R2*R3+R4
```

$n = 16$ is used, `R1` contains the A_i value, `R5` = 0xffff is used to get the n (16) least significant bits of A_i, and `R2` contains the constant $\mathcal{M}$. `MLA` is the multiply-accumulate instruction. Though the program is very simple, it requires a multiplication unit. Furthermore, in order to get 32-bit data, the above procedure should be done twice. Some processors provide 'multiply long' instruction for 32-bit multiplication, which results in 64-bit output. If the multiplication unit is not in the data-path, the program might become complicated with software implementation of the multiplications algorithm.

Another generator that we want to consider is a type of additive generator devised by G. J. Mitchell and D. P. Moore [20] and defined by

$$X_n \;=\; (X_{n-24} \,+\, X_{n-55}) \bmod m, \quad n \geq 55. \qquad (4)$$

This is also said to form a *lagged Fibonacci sequence* (LFS). The following piece of code is a version optimized for performance concerns.

```
LDR   R5, [R0], #4 ; R5=mem[R0], R0+=4
LDR   R6, [R1], #4 ; R6=mem[R1], R1+=4
ADD   R5, R5, R6   ; R5+=R6
STR   R5, [R2], #4 ; mem[R2]=R5, R2+=4
AND   R0, R0, R3   ;
AND   R1, R1, R3   ;
AND   R2, R2, R3   ;
```

`R0`, `R1` and `R2` point to the $n-24$, $n-55$ and n-th elements respectively in a cyclic list, and `R5` contains the X_n value after the run. Although the size of the cyclic list required by the generator is 55, we implemented it on 64 for simplicity and performance. It is assumed that the list starts at the address that has 0s in the 8 least significant bits. The last 3 `AND` instructions are used to clear the address bit for 2^8 and hence to keep the addresses from going over the boundary of the cyclic list. The addresses increase by 4 because one word (32-bit long) is made up of 4 bytes. The merits of this generator are speed, good random characteristics and not requiring a multiplication unit. It is also scalable on the bit length.

4. SELECTIVE-RUN OF RANDOM VECTORS

In this section, we propose a more novel testing technique, in which an embedded processor is used to run the preselected patterns through fault simulation.

Let a core require P words to fully load a test vector including its inputs and scan cells. For a random sequence $\langle X_n \rangle$, the sequence of random test patterns can be represented by

$$\langle V_n \rangle \;=\; \langle (X_{ns},\, X_{ns+1},\, ...,\, X_{ns+P-1}) \rangle \qquad (5)$$

BIST circuits usually apply test vectors at regular intervals of s, and generally $s = P$ because they cannot use memory to store the random numbers. The processor-based BIST technique can reuse the previously generated random numbers as part of the current and the following vectors by storing them in the processor memory. Hence some numbers in the sequence, $\langle V_n \rangle$, reappear in the following tuples when $s < P$. Because the required memory space to store a test vector is so small (P words), it does not incur any meaningful cost to the test of SOCs. Let us assume that 60 words are needed to apply a test vector to a core. Currently, many BIST algorithms generate 60 new words for every vector. In processor-based BIST, we can utilize the processor memory to store these 60 words. For the next random test vector, we can generate n new words and simply reuse $60 - n$ words stored in the memory. If $n = 1$, then the test generation time can be reduced up to 1/60.

As the number of test vectors increases, i.e., n becomes larger, it tends to be harder to detect additional faults due to the so called random pattern resistant structures in the circuit under test. Hence during random pattern testing, most of the patterns are not used for additional fault detection,

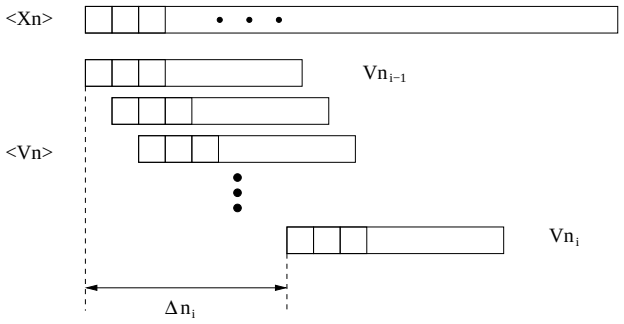

Figure 3: Random Numbers and Test Vectors

Core No.	Distance Δn_i
0010	100010011101
1000	000001010100
1000	000000000100
0000	000000000110
...	...

Figure 4: Encoded Data Format

and only a small number contributes to the fault coverage increase. Let faults be additionally detected when $n = n_i$, where i is a sequentially indexed number every time additional undetected faults are detected. We can apply test vectors only when $n = n_i$ instead of every n, thereby reducing the number of test applications. We encode $\Delta n_i = n_i - n_{i-1}$, so that the processor can apply the next test pattern after it simply generates Δn_i random numbers. Figure 3 shows the relations between the sequences $\langle X_n \rangle$ and $\langle V_n \rangle$. $\langle V_n \rangle$ is a sequence of P consecutive numbers from $\langle X_n \rangle$. Through fault simulation, those vectors that detect additional faults are extracted and encoded with the distances from the preceding vectors. The circuit under test is only exposed to those vectors applied by the processor. For optimal performance, testing is composed of two phases. During the first phase, every random pattern is just applied to the cores to detect easy faults, and then during the next phase, only the selected random patterns are applied, which reduces the whole testing time.

An acceleration method that can be used in processor-based BIST schemes was proposed in [11]. The method searches for a vector that maximizes the fault coverage within the small window boundary that is confined by the margin on the length of a test vector. This restricts the size of the search space too much, and the method is often forced to use unnecessary test vectors for fault coverage. In contrast, the method in this paper searches the whole pattern sequence to find those vectors that detect the undetected faults, so it does not waste time by applying unnecessary test vectors, and hence minimizes the testing time.

This technique is not only capable of testing each core in one run but also of testing every core in the same run. If we select test patterns from the same random number generator, it is possible to test every core in a same run by using the data format in Figure 4 for Δn_i. This format eliminates the need to separately generate the random numbers for each core. As all the cores share the generated test patterns, the test generation time is greatly reduced. The first field indicates the core that is going to be tested and the next field shows how many random numbers should be generated before the test application. Thus the first entry means that after 2,205 random number generation, the test pattern will be applied to the core No. 2. The next entry shows that the core No. 8 will be exposed to the test pattern after 84 random numbers, etc.. We used 12 bits for the distance field, which was justified using the experiments in the following section where the distance record showed that over 97 % of the distances are less than 4,096. For those cases of distances over a 12-bit boundary, we reserved '0000' in the 'Core No.' field of Figure 4. If an entry has '0000' in the field, the distance is calculated by multiplying the 'distance' field value with 2^{12}. This is combined with the next entry to give a range up to 2^{24}. The fourth entry in Figure 4 implies that random numbers are to be generated 6×2^{12} times.

In processor-based BIST, the testing time is mostly the sum of test generation time and test application time.

$$T = G + A \tag{6}$$

The total testing time for the individual runs can be represented by the summation of the individual core testing time.

$$T_I = \sum_i G_i + \sum_i A_i \tag{7}$$

The testing time for the shared run using the data format of Figure 4 can be represented as follows.

$$T_S = Max(G_i) + \sum_i A_i \tag{8}$$

We will see the speedup of the shared run in the following section.

Let the number of test vectors be N to achieve a targeted fault coverage in random BIST. Then a hardware-based BIST technique requires

$$\#\text{ of Generations} = N \times P \tag{9}$$

$$\#\text{ of Applications} = N \tag{10}$$

and our technique requires

$$\#\text{ of Generations} = N \tag{11}$$

$$\#\text{ of Applications} \ll N. \tag{12}$$

5. EXPERIMENTAL RESULTS

In order to evaluate the various PRNGs, fault simulation experiments were conducted on ISCAS benchmark circuits. The results of the fault simulations are shown in Table 2. Each entry in the table gives the fault coverage for all the nonredundant faults in a circuit after applying 32,000 test vectors. 'Reuse' column of Table 2 includes the fault coverages from the technique that reduces the test generation time by reusing the random numbers.

The two cellular automata show lower coverage on circuits 's1494', 's15850' and 's38417'. 'LFS' and 'Reuse' show good performance even compared to the ABIST method. Furthermore, since the 'Reuse' technique shows a performance no worse than the others, it is better to use this technique considering its potential speedup.

In the experiments of the selective-run, 1,000 random test vectors were applied onto each benchmark circuit during the first phase, and using a fault simulator, we selected those vectors that could detect additional faults and applied them

Table 2: Fault Coverage for Benchmark Circuits (%)

Circuit	CA		ABIST	LFS	Reuse
	CA1	CA2	n=16		s=1
c2670	91.44	88.11	87.42	92.49	88.39
c3540	99.36	96.86	99.67	99.60	99.70
c5315	99.89	99.96	100	100	100
s1238	96.35	96.27	99.53	99.84	99.14
s1423	99.53	99.33	99.87	99.87	100
s1494	76.77	76.77	100	100	100
s5378	97.54	96.14	99.73	99.67	99.84
s9234	92.45	93.47	93.47	93.00	92.68
s13207	99.35	99.41	98.25	98.11	98.97
s15850	91.37	91.72	95.47	95.80	95.70
s35932	99.95	99.92	100	100	100
s38417	94.70	92.31	95.42	95.42	94.97
s38584	99.44	99.49	99.38	99.49	99.48

Table 3: Simulation Results of Selective-Run

Circuit	FC w/ 1000	Last FC	No. Reuse	Sel. Run	
				No.	Speedup
c2670	84.98	99.99	1,820,625	1,071	1,699.93
c3540	84.02	100	90,899	1,140	79.74
c5315	99.56	100	6,044	1,020	5.93
s1238	89.55	100	274,720	1,072	256.27
s1423	97.32	100	68,272	1,018	67.06
s1494	97.44	100	5,998	1,025	5.85
s5378	95.36	100	63,878	1,161	55.02
s9234	80.08	99.57	1,438,429	1,410	1,020.16
s13207	85.89	100	231,207	1,449	159.56
s15850	89.42	99.4	1,813,420	1,420	1,277.06
s38417	90.73	99.92	1,825,131	2,141	852.47
s38584	93.07	99.95	1,427,330	1,688	845.57

during the next phase. For a pseudo-random number generator, we used a *lagged Fibonacci sequence* (LFS) which is defined by Equation 4. We termed the LFS method with $s = 1$ in Equation 5 as 'Reuse' for this testing. The second column in Table 3 shows fault coverage for each circuit after applying 1,000 test vectors. The third column shows the fault coverage after applying all the test patterns in the fourth column when the 'Reuse' method is used. Even though the required number of random vectors to reach the last fault coverage is generally big, the number of vectors that contribute to the fault coverage is very small as in the 'No.' column of the 'Sel. Run' block. Each number shows the total number of test vectors to be applied onto each circuit using our technique including the first 1,000 random vectors. The random number generation time is much smaller than the test application time for each core. Note that in the 'Reuse' technique, each number in the column is the number of random numbers to generate as well as the number of test patterns to apply to the circuit. In hardware-based BIST technique, the required number of random numbers to generate is P times bigger than each number in the column as shown in Equation 9. The last column shows the speedup of the selective-run technique over the 'Reuse' in test application time. The speedup was calculated as follows.

$$Speedup = \frac{\text{Test Application Time of 'Reuse'}}{\text{Test Application Time of Selective-Run}} \quad (13)$$

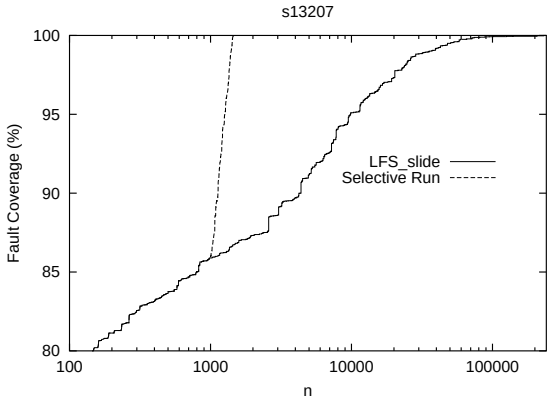

Figure 5: Trace of Fault Coverage of s13207

Table 4: Data Reduction of Selective-Run

Circuit	Deterministic		Golomb	Selective-Run		
	Vec.	Words	Words	Vec.	Words	% Red.
c2670	112	560	137	1	41	92.68
c3540	101	202	131	–	70	65.35
c5315	19	114	38	–	10	91.23
s1238	70	140	73	–	36	74.29
s1423	18	54	26	–	9	83.33
s1494	23	23	25	–	13	43.48
s5378	156	1,092	213	–	81	92.58
s9234	398	3,184	482	17	341	89.29
s13207	327	7,194	674	–	225	96.87
s15850	439	8,780	784	50	1,210	86.22
s38417	1,053	54,756	3,433	39	2,599	95.25
s38584	585	26,910	1,513	11	850	96.84

The results show that very big speedups are achieved on most circuits because the selected patterns are only a small portion of the total patterns.

Figure 5 shows the graph of fault coverage versus n for the benchmark circuit 's13207'. Our *selective-run* technique significantly reduces the test application time. Note that n is in logarithmic scale.

The test generation time is also reduced by using the format of Figure 4. The speedup of the shared run compared to the individual run in test generation time is 4.97 times in this experiment.

We next compare our technique with the deterministic method to see the data reduction. Many BIST techniques use a combination of BIST and deterministic test. After running random BIST testing, they usually apply deterministic test patterns from ATPG tools in order to achieve the targeted fault coverage. In Table 4, we assumed that 1,000 test vectors have been applied using random BIST techniques such as LFS, and then we compared the required data sizes of the two techniques to reach 100% fault coverage. The second column (Vec.) shows the number of test vectors for each circuit to reach 100% fault coverage after applying 1,000 random test patterns. They were obtained

using a commercial ATPG tool. The 'Vec.' column in the *Selective-Run* block shows the number of vectors that are additionally needed after applying our technique because of the limitation of random testing. [4] showed that Golomb coding is very efficient to compress test data. We performed the technique on the deterministic test data set and found that $m = 32$ gave the best overall performance. The third, fourth and sixth columns ('Words') show how many words are needed in each technique. The table shows that the selective-run technique is more efficient in data compression performance than the Golomb coding technique on most circuits. Note that the selective-run technique does not use any compression algorithm, and hence its implementation is straightforward. The last column shows the percentage reduction of data size, which is computed as follows.

$$\frac{\text{Deterministic Data} - \text{Selective-Run Data}}{\text{Deterministic Data}} \times 100 \qquad (14)$$

As seen in the '% Red.' column, our technique requires a very small amount of memory compared to the traditional methodology. The percentage reductions show that very high compressions are achieved over 90% on most circuits. In processor-based testing, the memory requirement is also an important factor in order to use fast internal memories such as cache.

6. CONCLUSIONS

Several pseudo-random number generators have been implemented on an embedded processor and evaluated through benchmark circuits. A novel selective-random test technique has been proposed. Since the technique only applies those vectors that contribute to the fault coverage increase, it does not waste testing time running unnecessary vectors. The test application time was shown to be greatly reduced with a speedup of more than several hundreds on most benchmark circuits. Our technique also eliminates the requirement of a large data memory for the test vectors, because self-producing pseudo-random number generators are used to construct deterministic test patterns. The required data sizes were only a fraction of those required in traditional BIST schemes. This selective-run technique and the reuse of the generated random numbers can be easily implemented on a processor with a small amount of code and data memory.

7. REFERENCES

[1] Y. Zorian, E. J. Marinissen, and S. Dey. Testing embedded-core-based system chips. In *Proceedings of the International Test Conference*, pages 130–143, 1998.

[2] M. Ishida, D. S. Ha, and T. Yamaguchi. COMPACT: A hybrid method for compressing test data. In *Proceedings of the VLSI Test Symposium*, pages 62–69, 1998.

[3] T. Yamaguchi, M. Tilgner, M. Ishida, and D. S. Ha. An efficient method for compressing test data. In *Proceedings of the International Test Conference*, pages 79–88, 1997.

[4] A. Chandra and K. Chakrabarty. System-on-a-chip test-data compression and decompression architectures based on golomb codes. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 355–368, March 2001.

[5] A. Jas and N. A. Touba. Test vector decompression via cyclic scan chains and its application to testing core-based designs. In *Proceedings of the International Test Conference*, pages 458–464, November 1998.

[6] N. A. Touba and E. J. McCluskey. Bit-fixing in pseudorandom sequences for scan BIST. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 545–555, April 2001.

[7] G. Mrugalski, J. Tyszer, and J. Rajski. Synthesis of pattern generators based on cellular automata with phase shifters. In *Proceedings of the International Test Conference*, pages 368–377, 1999.

[8] S. Hellebrand, H.-J. Wunderlich, and A. Hertwig. Mixed-mode BIST using embedded processors. In *Proceedings of the International Test Conference*, pages 195–204, 1996.

[9] J. Rajski and J. Tyszer. *Arithmetic Built-In Self-Test: For Embedded Systems*. Prentice Hall, 1998. ISBN:0137564384.

[10] S. Hwang and J. A. Abraham. Reuse of addressable system bus for SOC testing. In *IEEE International ASIC/SOC Conference*, September 2001.

[11] S. Hwang. Processor based built-in self-test for embedded cores. Technical Report UT-CERC-TR-JAA-01-5, Computer Engineering Research Center, Univ. Texas, Austin, TX, 2001.

[12] K. Radecka, Janusz Rajski, and Jerzy Tyszer. Arithmetic built-in self-test for DSP cores. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1358–1369, November 1997.

[13] C. A. Papachristou, F. Martin, and M. Nourani. Microprocessor based testing for core-based system on chip. In *Proceedings of Design Automation Conference*, pages 586–591, 1999.

[14] S. M. Thatte and J. A. Abraham. Test generation for microprocessors. *IEEE Transactions on Computers*, pages 429–441, June 1980.

[15] R. S. Tupuri and J. A. Abraham. A novel functional test generation method for processors. In *Proceedings of the International Test Conference*, pages 743–752, November 1997.

[16] J. Shen and J. A. Abraham. Native mode functional test generation for processors with applications to self test and design validation. In *Proceedings of the International Test Conference*, pages 990–999, October 1998.

[17] J. Rajski, G. Mrugalski, and J. Tyszer. Comparative study CA-based PRPGs and LFSRs with phase shifters. In *Proceedings of the VLSI Test Symposium*, pages 236–245, 1999.

[18] S. Wolfram. Statistical mechanics of cellular automata. *Reviews of Modern Physics*, 55:601–644, July 1983.

[19] P. D. Hortensius, H. C. Card, and R. D. McLeod. Parallel random number generation for VLSI using cellular automata. *IEEE Transactions on Computers*, 38:1466–1473, October 1989.

[20] D. E. Knuth. *The Art of Compuer Programming*, volume 2. Addison-Wesley, Reading, MA, 3rd edition, 1997. ISBN:0201896842.