Compressing Functional Tests for Microprocessors

Kedarnath J. Balakrishnan
NEC Labs. America
4 Independence Way, Suite 200
Princeton, NJ 08540, USA
bala@nec-labs.com

Nur A. Touba
Computer Engineering Research Center
University of Texas at Austin
Austin, TX 78712, USA
touba@ece.utexas.edu

Srinivas Patil
Intel Corporation
1501 S. Mopac Suite 400
Austin, TX 78704, USA
srinivas.patil@intel.com

Abstract

In the past, test data volume reduction techniques have concentrated heavily on scan test data content. However, functional vectors continue to be utilized because they target unique defects and failure modes. Hence, functional vector compression can help alleviate the cost of functional test. Scan vector compression techniques are generally unsuitable in the functional domain and techniques specially tailored for functional test compression are required. Additionally, it may be possible to perform compression and decompression using software techniques without incurring the overhead of dedicated hardware. This paper proposes a set of software techniques targeted towards functional test compression.

1. Introduction

As circuit sizes scale with technology, manufacturing test cost is contributing a larger share of the total cost of manufacturing a chip [18]. One of the key contributing factors to test cost is test time and test data volume. Since a key measure of cost efficiency of a manufacturing test process is tester throughput, it is imperative that we reduce the time a chip spends at the tester. Most of the test time is spent loading the test data through a slow tester interface, though the design is typically capable of supporting a higher tester throughput. However, tester throughput cannot be increased without a proportional increase in the cost of the tester itself (which usually runs in the millions). Similarly, testers have a limited amount of memory available to store test vectors. Test data could be partitioned and then reloaded into tester memory one partition at a time, but tester reloads cost time as well. Increasing the tester memory is not a good option due to cost considerations. Thus, techniques are necessary to reduce the Test Application Time (TAT) and Test Data Volume (TDV).

Due to the criticality of TAT and TDV issues, there has been abundant research in the area of scan test data compression [3]-[25]. Test data compression techniques primarily try to reduce the test data volume but many techniques try to reduce the test time as well. For scan vectors, the test data volume is dominated by the data required to load and unload the scan chains (typically 3 bits per scan cell - 1 bit for stimulus, 1 bit for observation and 1 extra bit to specify which scan cells need to be masked out during observation).

A variety of test data compression techniques targeted towards scan vectors have been proposed in the literature. Some of them utilize general-purpose techniques such as *run-length encoding (RLE)* [7][12][13]. RLE relies on representing a repetition of a certain symbol (0 or 1) with a codeword which denotes the count of the repeated symbol and the symbol itself. Golomb Codes are one such example of RLE. Techniques utilizing LZW algorithm (the UNIX "compress" and "gzip" utilities use a version of this algorithm) have also been proposed [25].

The other set of techniques rely on utilizing linear networks which have been widespread in the built-in self-test (BIST) domain [2]. Such structures include linear feedback shift registers (LFSRs) [3] and phase shifters. Phase shifters are XOR networks used to reduce the shift dependency amongst bit streams from different stages of an LFSR. XOR networks (which include phase shifters) can be thought of as linear networks in the space domain whereas LFSRs can be thought of as linear networks in the time domain. Thus, the time behavior of an LFSR can be translated into an equivalent linear combinational network consisting of XORs [14]. Linear networks are popular because their behavior is well-understood and are amenable to mathematical rigor.

Despite a body of research in the area of scan test data compression, there has been very little work in the area of functional test compression. It should be noted that software compression has been explored in the area of embedded applications [26][27], where the code size can impact utilization of on-chip resources needed to store firmware and software programs. These techniques also rely on dedicated on-chip hardware for decompression unlike the software-based techniques proposed in this paper which entail no hardware overhead.

In this paper, we propose compression techniques specially suited to a functional test environment. The compression and decompression is performed using software routines and without support from any dedicated compression hardware. The proposed compression techniques can be used to compress either deterministic functional tests or microprocessor self-test programs that utilize random instructions (e.g., [28], [31]-[35]). Hence, the proposed techniques are complementary to existing techniques and can be used in

Proceedings of the 14th Asian Test Symposium (ATS '05)
1081-7735/05 $20.00 © 2005 IEEE

conjunction with them to reduce TAT and TDV. The remainder of the paper is organized as follows: Section 2 details the underlying framework for the compression scheme. Section 3 discusses how to partition the functional tests into blocks and order them to minimize test time. Section 4 lists the compression methods used in this study followed by Sections 5 and 6 which provide the experimental results and conclusions respectively.

2. Functional Test Framework

As shown in Fig. 1, we assume the functional test patterns are applied using a low-cost tester (generally a structural tester) with limited pin access, similar to the scheme described in [28]. For the moment, assume the vectors are not compressed. The test vectors are loaded into the cache using an interface which is running at a speed slower than the core clock frequency, usually at the front side bus (FSB) frequency. The functional tests then execute from the cache at the core clock frequency. At the end of the test session, the results are deposited back in the cache and subsequently downloaded by the tester. Thus, the speed of the tester-die interface (the FSB) and the size of the cache determine the test time and the code size of the vectors which can be applied in a test session. The impact of the slow tester interface and cache size on test time and volume can be reduced greatly if the functional tests are loaded in compressed form. The decompression of the tests takes place on-chip at the core clock frequency with very little impact on test time. Further, if the decompression is done on-the-fly and piece-meal as the vectors are needed, the cache can be utilized more effectively. This technique effectively increases the virtual size of the cache and allows larger programs to be executed.

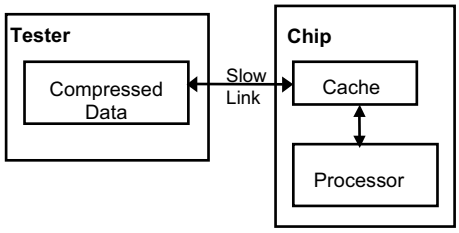

Figure 1. Test Application Framework

Figure 2 shows the cache image under the proposed scheme. The data loaded from the tester includes two components: a preamble which contains the compression and decompression routines followed by the main portion which contains the compressed test data. Segments of the cache are reserved to store the decompressed instruction/code streams, the uncompressed response and the compressed response.

The execution flow is shown in Figure 3: a decompression routine is executed to decompress a block of test data and the resultant decompressed instruction stream is stored in a reserved portion of the cache. Control is now transferred from the decompression routine to the decompressed instruction stream. As a result of the execution of the instruction stream, response data is now deposited back in a reserved area of the cache and control is now transferred to a compression routine to compress the response data. The compressed data may either be transferred immediately back to the tester or stored in a reserved area for transfer later.

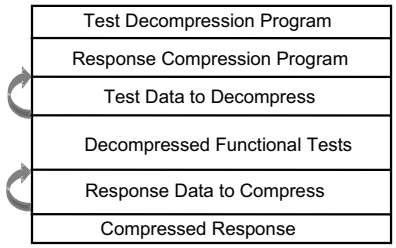

Figure 2. Cache Image

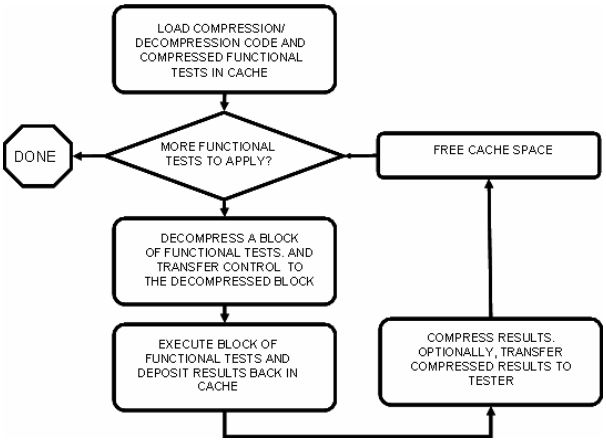

Figure 3. Execution Flow

The portion of the cache containing the decompressed instructions and uncompressed response can now be freed. Additionally, if the compressed response data has been transferred to the tester, that area can be reclaimed as well. Control now returns to the decompression program where the same process is repeated for another block of data. It should also be noted that the tester operations and the operations internal to the core (compression/ decompression/execution) can be overlapped to hide some of the latencies. This is explained further in Sec. 3.

From the discussion above, it is imperative that the compression and decompression routine themselves should occupy the minimum amount of space in the cache. Additionally, the execution time for compression and decompression should be insignificant compared to the execution time for the test program itself. Based on this framework, we discuss different compression and decompression schemes in Sec. 4.

3. Partitioning and Ordering Functional Test Blocks

In order to overlap the time required to transfer the compressed data from the tester to the cache with the time required to decompress and execute the tests, the functional tests are partitioned into blocks of instructions with as fine granularity as possible while allowing each block to be executed independently of the others. How small or large each block is will depend on the overall structure of the tests. The key is that each block of functional tests can be decompressed and executed independent of the others.

For each block *i*: let $transfer_i$ be the time required to transfer the compressed block from the tester to the cache; let $apply_i$ be the time to apply the test which involves decompressing the block, executing the block, and compressing the response; and, let $memory_i$ represent the amount of memory in the cache required for applying the block which includes both the memory for storing the uncompressed code as well as the uncompressed response. Then it is possible to overlap the time for transferring block *i* from the tester, $transfer_i$, with the time for applying the previous *b* blocks $\sum_{i-b}^{i-1} apply_k$ provided there is sufficient memory in the cache for $\sum_{i-b}^{i-1} memory_k$.

The blocks should be optimally ordered to minimize the overall test time. If the sum of the tester transfer time for all the blocks $\sum_1^n transfer_k$ exceeds the sum of the time to apply the blocks $\sum_1^n apply_k$ then the tester transfer time is the bottleneck. In that case, the blocks should be ordered so that the tester is constantly transferring to the cache, and the block with the shortest application time should be ordered last to minimize the delay beyond the last transfer. However, if the sum of the application time for all the blocks $\sum_1^n apply_k$ exceeds the sum of the time to transfer the blocks $\sum_1^n transfer_k$ then the application time is the bottleneck. In that case, the blocks should be ordered so that the block with the shortest transfer time is transferred first (to minimize the latency before the microprocessor begins execution), and the remaining blocks should be ordered so that the microprocessor is never waiting for a block to be transferred.

How easily the blocks can be ordered to minimize test time will depend on how large the blocks are and how much cache memory is available. The more blocks that can be stored in cache memory at one time, the easier it is to ensure that the bottleneck, whether tester transfer time or test application time, is not stalled at any point.

In the next section, different encoding schemes will be described which offer different tradeoffs between the amount of cache memory required by the decompression program versus the amount of compression that is provided. If the tester transfer time is the bottleneck, then encoding schemes that provide greater compression (thereby reducing the amount of data the tester has to transfer) may be preferable. However, if the limited cache memory is resulting in a suboptimal ordering of the blocks, then encoding schemes that use a smaller decompression program and dictionary may be preferable.

4. Compression Methods

In this section, different compression schemes are studied that could be suitable for functional test compression. One main requirement of the functional test framework is that the decompression should be done fully in software. The decompression program size adds to the compressed data and hence the program size should be minimized as much as possible. Similar to scan data compression, the response of functional tests can be compressed using lossy compression techniques. However, the input tests need to be compressed in a lossless manner. Functional tests are programs which mean they are fully specified unlike scan vectors that can have unspecified bits. Three different compression schemes have been investigated as part of this work. They are discussed in detail in the following sections.

4.1 Run-Length Encoding

Run length encoding involves replacing the runs (or repetitions) of different characters with fixed size codewords. It is very effective in data sets where one or more characters are repeated a lot times together. The decompression program is very simple and can be implemented with a relatively small number of instructions in software.

1	→ 000	00001	→ 100
01	→ 001	000001	→ 101
001	→ 010	0000001	→ 110
0001	→ 011	00000001	→ 111

Figure 4. 3-bit run length code for runs of 0's

Figure 4 illustrates a 3-bit run length code. The data appearing on the left side is coded with the codewords given on the right hand side. In this code, runs of 0's are compressed. Another option is to use an alternating run-length code where both runs of 0s and runs of 1's are compressed.

4.2 Huffman Codes

Huffman coding [29] is a fixed to variable code. The idea is to encode fixed size blocks of input data to variable size codewords based on the frequency of oc-

currence of the blocks. The blocks that appear more frequently are encoded with fewer bits. This is illustrated in Fig. 5 that shows blocks and their frequency of occurrence and their corresponding Huffman codes.

Block	Frequency	Code
000	15	1
001	12	10
010	10	1000
011	9	1001
100	8	1010
101	8	11110
110	5	111110
111	1	111111

Figure 5. Example of Huffman code

4.3 Liv-Zempel-Welch (LZW)

LZW codes fall under the larger category of sliding window dictionary based Lempel-Ziv algorithms [30]. In the LZW variant, the dictionary is created dynamically. If the symbol is not present in the dictionary, it is added. The old entries are automatically removed when the window is filled.

4.4 Operand Factorization

The three compression techniques discussed above are general compression techniques that can be used for any data set. Hence, schemes that utilize the nature of the data to be compressed to further increase the compression maybe useful. Functional tests are programs that are run on the microprocessor. The programs or "code" are a series of instructions in the instruction set format of the microprocessor. The instructions can be said to roughly consist of operations that are performed on a set of data. The basic idea of operand factorization [26] is to separate out the operation part of the instruction (opcodes) and the operand part of the instruction (registers and immediates). The two parts are then compressed separately. During decompression, they are put together to form the original program. The main advantage of this type of partitioning is that the two parts will have more repetitions and hence more compression can be achieved. On the other hand, the decompression process will be a little more complicated than decompressing a single stream of compressed data. In this work, experiments have been performed with operand factorization to see how much improvement it provides.

5. Experimental Results

The compression schemes discussed in the previous sections were implemented. Experiments were performed on the SPEC2000 benchmark programs compiled for an ARM microprocessor. Table 1 shows the size of the decompression program in bytes that is required for each of the compression schemes: run-length coding, Huffman coding, and LZW. Three different types of run-length coding were used: 2-bit codewords encoding runs of 0's, 2-bit alternating codewords encoding both runs of 0's and 1's, and 3-bit codewords encoding runs of 0's. For Huffman coding, three different symbol lengths were used: 8 bit, 10 bit and 12 bit. For LZW, three different window sizes were used: 256 bytes, 1024 bytes, and 4096 bytes. The results in Table 1 show the total amount of memory (in bytes) required for decompression with a breakdown of how much is for the decompression program and how much is for the dictionary. As can be seen in Table 1, run-length coding requires much less memory for decompression in comparison to the other two. Huffman coding is in the middle, and LZW requires the most amount of memory for decompression.

Table 1. Memory (in bytes) required for decompression

Method	Type	Code Size	Dictionary Size	Total
RLE	2 bit	96	0	96
	alt. 2 bit	128	0	128
	3 bit	160	0	160
Huffman	8 bit	832	256	1088
	10 bit	832	1024	1856
	12 bit	832	4096	4928
LZW	256	8172	256	8428
	1024	8172	1024	9196
	4096	8172	4096	12268

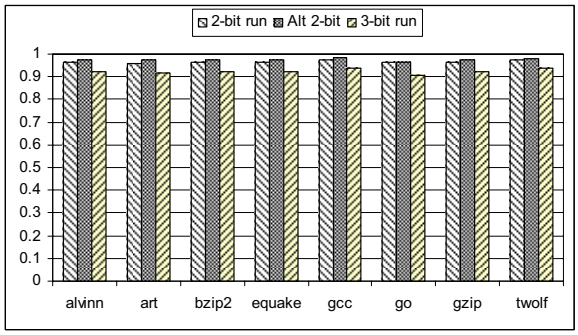

Figure 6. Compression obtained with run-length code

Figure 6 shows the amount of compression obtained when using three different run-length codes on the SPEC2000 benchmarks compiled for ARM. The *y*-axis is the compression ratio defined as *(compressed data)/(original data)*. As can be seen in Figure 6, run-length coding does not provide much compression. The alternating run-length code performed the worst. The 3-bit run length code performed the best. We tried 4-bit and longer run lengths codes, but the results were much worse. The conclusion that can be drawn from these results is that run-length codes, while requiring very little memory for decompression, are simply not effective for functional tests. There are not enough runs in the data to achieve significant compression.

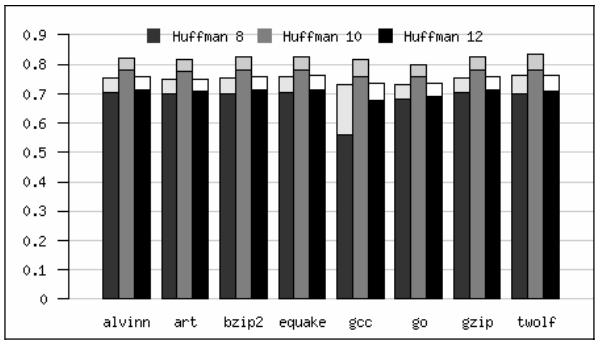

Figure 7. Compression obtained with Huffman coding

Figure 7 shows the amount of compression obtained when using Huffman coding with three different symbol lengths. For Huffman coding, performing operand factorization improves the compression. The full size of each bar is the compression without operand factorization, and the solid portion of each bar is the compression with operand factorization. Because some instructions occur more frequently than others, operand factorization is beneficial as it allows the Huffman code to better exploit this. As can be seen in Figure 7, having a symbol length of 8 outperforms symbol lengths of 10 and 12 even though it uses a smaller dictionary. The reason for this is that a symbol length of 8 aligns the symbols along word boundaries thus better exploiting correlations between instructions.

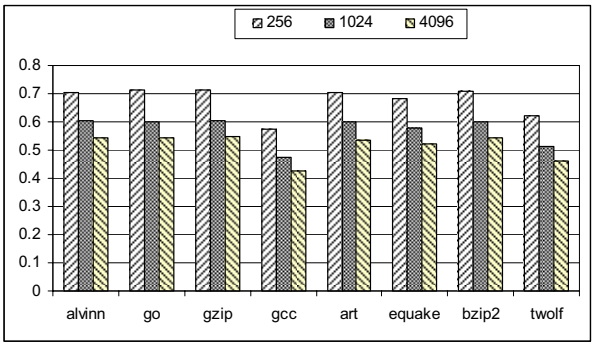

Figure 8. Compression obtained with LZW for ARM

Figure 8 shows the amount of compression obtained when using LZW with three different dictionary sizes for ARM code. The amount of compression monotonically improves with larger dictionaries. For a given application, the dictionary could simply be set as large as the available memory will allow. To get an idea how much the results varied for different instruction sets, we also generated results for x86 code which is shown in Figure 9. The results and trends are similar. To get an idea of the inherent compressibility of the ARM code, we calculated the entropy for each of the benchmark programs for a symbol size of 32 to find the theoretical limit on the amount that the code could be compressed. This is shown in Table 2 along with the best compression that we achieved for each of the different coding schemes. As can be seen in Table 2, LZW gets fairly close to the entropy limit.

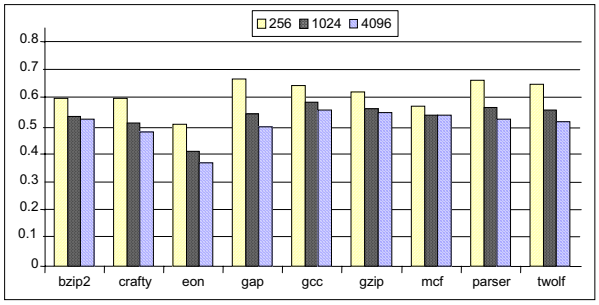

Figure 9. Compression obtained with LZW for x86

Table 2. Comparison with entropy limit

Benchmark	RLE	Huffman	LZW	Entropy
alvinn	0.920	0.706	0.542	0.395
art	0.919	0.698	0.534	0.391
bzip2	0.923	0.702	0.542	0.396
equake	0.923	0.706	0.520	0.393
gcc	0.936	0.559	0.427	0.372
go	0.909	0.681	0.541	0.392
gzip	0.923	0.702	0.546	0.396
twolf	0.939	0.697	0.461	0.385

6. Conclusions

This paper has presented a methodology for compressing functional tests. The decompression is performed in software requiring no additional hardware. It can be used to compress both deterministic functional tests as well as self-test programs. Experimental results show that while run-length codes require very small decompression programs, they provide very little compression for fully specified functional tests. Huffman coding where the symbol length aligns with word boundaries provides a modest amount of compression. Operand factorization can be used in conjunction with Huffman codes to further improve the compression (though not dramatically). LZW provides the most compression, but also requires the largest decompression program. Depending on the available memory for a particular application, an appropriate coding technique can be selected.

Acknowledgements

This material is based on work supported in part by the Intel Corporation and in part by the National Science Foundation under Grant No.CCR-0306238.

References

[1] C. L. Chen, “Linear Dependencies in Linear Feedback Shift Registers”, *IEEE Transactions on Computers*, Vol. C-35, No. 12, pp. 1086-1088, Dec. 1986.

[2] P. H. Bardell, W. H. McAnney and J. Savir, *Built-In Test for VLSI: Pseudorandom Techniques,* John Wiley and Sons, 1987.

[3] B. Koenemann, "LFSR-Coded Test Patterns for Scan Designs", *Proc. European Test Conference*, 1991.

[4] I. Pomeranz, L. Reddy and S. M. Reddy, "COMPACTEST: A Method to Compact Test Sets for Combinational Circuits", *IEEE Trans. CAD*, Vol. 12, No. 7, pp. 1040-1049, July 1993..

[5] S. Bhatia and P. Varma, "Test Compaction in a Parallel Access Scan Environment", *Proc. Asian Test Symposium*, pp. 300-305, 1997.

[6] K. Chakrabarty, B. T. Murray, J. Liu and M. Zhu, "Test Width Compression for Built-In Self Testing", *Proc. Int. Test Conference*, pp. 328-337, 1997.

[7] A. Jas and N. A. Touba, "Test Vector Decompression Via Cyclical Scan Chains and Its Application to Testing Core-Based Designs", *Proc. Int. Test Conf.*, pp. 458-464, 1998.

[8] I. Hamzaoglu and J. Patel, "Reducing Test Application Time for Full Scan Embedded Cores", *Proc. Fault Tolerant Computing Symposium*, pp. 260-267, 1999.

[9] A. Jas, J. Ghosh-Dastidar and N. A, Touba, "Scan Vector Compression/Decompression Using Statistical Coding," *Proc. VLSI Test Symposium,* pp. 25-29, 1999.

[10] A. Jas, B. Pouya and N. A. Touba, "Virtual Scan Chains: A Means for Reducing Scan Length in Cores", *Proc. International Test Conference*, pp. 73-78, 2000.

[11] D. Das and N. A. Touba, "Reducing Test Data Volume Using External/LBIST Hybrid Test Patterns", *Proc. International Test Conference*, pp. 115-122, 2000.

[12] A. Chandra and K. Chakrabarty, "System-on-a-Chip Test Data Compression and Decompression Architectures Based on Golomb Codes", *IEEE Trans. CAD*, Vol. 20, No. 3, pp. 355-368, 2001.

[13] A. Chandra and K. Chakrabarty, "Frequency-directed Run-length (FDR) Codes with Application to System-on-a-chip Test Data Compression", *Proc. VLSI Test Symposium*, pp. 42-47, 2001.

[14] I. Bayraktaroglu and A. Orailoglu, "Test Volume and Application Time Reduction Through Scan Chain Concealment", *Proc. DAC.*, pp. 151-155, 2001.

[15] F. Hsu, K. Butler and J. Patel, "A Case Study on the Implementation of the Illinois Scan Architecture", *Proc. Intl. Test Conf.*, pp. 538-547, 2001.

[16] C. Barnhart, V. Brunkhorst, F. Distler, O. Farnsworth, B. Keller and B. Koenemann, "OPMISR: The Foundation for Compressed ATPG Vectors", *Proc. International Test Conference*, pp. 748-757, Oct. 2001.

[17] C. V. Krishna, A. Jas and N. A. Touba, "Test Vector Encoding Using Partial LFSR Reseeding", *Proc. International Test Conference*, pp. 885-893, 2001.

[18] *International Technology Roadmap for Semiconductors (ITRS),* 2001 Edition, Test and Test Equipment Section.

[19] S. M. Reddy, K. Miyase, S. Kajihara and I. Pomeranz, "On Test Data Volume Reduction for Multiple Scan Chain Designs", *Proc. VTS.*, pp. 103-108, 2002.

[20] A. Khoche, E. H. Volkerink, J. Rivoir and S. Mitra, "Test Vector Compression Using EDA-ATE Synergies", *Proc. VLSI Test Symposium*, pp. 411-416, 2002.

[21] A. Chandra and K. Chakrabarty, "Reduction of SOC Test Data Volume, Scan Power and Testing Time Using Alternating Run-Length Codes", *Proc. Design Automation Conference*, pp. 673-678, 2002.

[22] E. H. Volkerink, A. Khoche and S. Mitra, "Packet-based input test data compression techniques", *Proc. International Test Conference,* pp. 154-163, 2002.

[23] J. Rajski *et al,* "Embedded Deterministic Test for Low Cost Manufacturing Test", *Proc. International Test Conference*, pp. 301-310, 2002.

[24] S. Mitra and Kee Sup Kim, "X-Compact: An efficient response compaction technique for test cost reduction", *Proc. Intl. Test Conf.,* pp. 311-320, 2002.

[25] F. Wolff and C. Papachristou, "Multiscan-Based Test Compression and Hardware Decompression Using LZ77", *Proc. Int. Test Conf.*, pp. 331-339, 2002.

[26] G. Araujo *et al*, "Code Compression Based on Operand Factorizatin", *Proc. 31st annual ACM/IEEE Intl. Symposium on microarchitecture*, pp. 194-201, 1998.

[27] J. Ernst *et al*, "Code Compression", *Proc. SIGPLAN conference on programming language design and implementation PLDI'97,* pp 358-365, 1997.

[28] P. Parvathala, K. Maneparambil and W. Lindsay, "FRITS - A Microprocessor Functional BIST Method", *Proc. Int. Test Conf.,* pp. 590-598, 2002.

[29] D. A. Huffman. "A method for the construction of minimum redundancy codes". *Proc. of the IRE*. Vol 40, Issue 9, pp. 1098-1101, Sept. 1952.

[30] A. Lempel and J. Ziv. "On the complexity of finite sequences". *IEEE Transactions on information theory*. Volume 22, Issue 1 pp. 75-81, Jan. 1976.

[31] J. Shen and J.A. Abraham, "Native mode functional test generation for processors with applications to self test and design validation," *Proc. International Test Conference*, pp. 990-999, 1998.

[32] K. Batcher and C. Papachristou, "Instruction randomization self test for processor cores," *Proc. VLSI Test Symposium*, pp. 34-40, 1999.

[33] C. Li and S. Dey, "Software-based self-testing methodology for processor cores," *IEEE Trans. on Computer-Aided Design*, Vol. 20, Issue 3, pp. 369-380, Mar. 2001.

[34] A. Kristic, W.-C. Lai, K.T. Cheng, L. Chen, and S. Dey, "Embedded software-based self-test for programmable core-based designs," *IEEE Design & Test of Computers*, Vol. 19, Issue 4, pp. 18-27, Jul. 2002.

[35] N. Kranitis, D. Gizopoulos, A. Paschalis, and Y. Zorian, "Instruction-based self-testing of processor cores," *Proc. of VLSI Test Symposium*, pp. 223-228, 2002.