Open access • Proceedings Article • DOI:10.1109/TEST.2002.1041756

Packet-based input test data compression techniques — Source link

E.H. Volkerink, Ajay Khoche, Subhasish Mitra

Institutions: Stanford University, Agilent Technologies, Intel

Published on: 07 Oct 2002 - International Test Conference

Topics: Test compression, Test data, Automatic test pattern generation, Huffman coding and Data compression

Packet-based Input Test Data Compression Techniques*

Erik H. Volkerink [1,2], Ajay Khoche [2], Subhasish Mitra [3]

[1]Center for Reliable Computing (CRC)
Stanford University, Stanford, CA

[2]Agilent Laboratories
Palo Alto, CA

[3] Intel Corporation
Sacramento, CA

Abstract

This paper presents a test input data compression technique, which can be used to reduce input test data volume, test time, and the number of required tester channels. The technique is based on grouping data packets and applying various binary encoding techniques, such as Huffman codes and Golomb-Rice codes. Experiments on actual industrial designs and benchmark circuits show an input vector data reduction ranging from 17x to 70x.

Keywords: Compression, RLE, Huffman, Golomb-Rice, LFSR, BIST, ATE, EDA.

1 Introduction

As the design complexity increases so does the test data volume [1, 2, 4]. This increase in test data volume results in the following critical problems:

- *Limited vector memory on ATE:* More test data means that the ATE will have to support more vector memory. However, depending on the requirements, such an ATE may not be available or may be very expensive. For this reason, vector sets are often truncated, resulting in a reduced product quality [2].
- *Long upload time:* Large test data volume can result in a long time to load/reload test vectors into the ATE memory, often ranging from several tens of minutes to hours [2, 5]. Since the ATE remains idle during the loading and reloading of test vectors, the ATE utilization is reduced. This can significantly increase the cost of test.
- *Limited I/O bandwidth:* The I/O bandwidth is defined as the number of input channels multiplied with the frequency capability of these channels. Even if the ATE has sufficient memory to store all the test data bits, the transmission of test data to and from the Device-Under-Test (DUT) can result in an unacceptable long test time due to the limited bandwidth between the ATE and the DUT. The bandwidth could be limited due to the limited channel frequency or due to a lack of a sufficient number of scan channels on the ATE or the DUT. In fact, in [4] it is shown that, with current test methods, test times will increase almost exponentially for the device complexity projections in the ITRS'99 roadmap [1].

Several techniques have been published to achieve test vector compression. Our technique is based on the packet matching compression technique described in [24]. This paper improves the technique significantly by introducing a compression algorithm based on encoding combinations of packets and applying different binary encoding techniques, like Huffman encoding and Golomb-Rice encoding.

The compression technique reduces the ATE vector memory requirements and upload time problems. Moreover, the technique can reduce the limited I/O bandwidth problem.

Experiments done on an industrial 7M gates network processor ASIC, show a reduction in input vector volume of about 55 times on top of the conventional static and dynamic compaction techniques [29].

This paper is organized as follows: Section 2 describes previous work on test data compression techniques. Section 3 presents the test data compression technique. Section 4 describes the experimental results. Section 5 describes several decompression implementations that can be used to retrieve the original data. Section 6 compares the technique with other published compression techniques. The paper concludes with the conclusions in Section 7.

2 Previous Work

Previous work on test data compression can be classified in the following categories:

- *Loss-less compression techniques [3, 5, 6, 9, 21]:* These techniques typically take advantage of certain sequences of bits in the vectors by encoding the sequences using a smaller sequence of bits.
- *Compression techniques based on LFSR reseeding [7, 10, 17, 19, 23]:* These techniques assume that a large proportion of the bits in the test vectors are unspecified (also called *don't care* bits). For every test vector, most of the techniques attempt to find one or multiple seeds for on-chip Linear Feedback Shift Registers (LFSRs) or XOR networks such that the bit sequence generated by the LFSRs or XOR-networks matches the test vector at the specified bit positions.

* This work was done at CRC, Stanford University

The LFSR size is significantly smaller compared to the number of scan flip-flops.

- *Pseudo Random BIST compression techniques [2,13,16]:* Pseudo-random (PR) vector generation using LFSRs belongs to this category. The PR vector generation techniques provide the best test data compression. However, to achieve similar fault coverage compared to external ATPG vectors, significantly more on-chip generated pseudo-random vectors are required. This potentially results in longer test times and power dissipation problems [4]. To reduce the number of required PR vectors, additional test points can be inserted. Moreover, additional external ATPG vectors (also called *top-off vectors*) can be applied after the PR vectors, to cover the faults that are hard to detect by the PR vectors [2]. To reduce the problem of long test times, deterministic BIST techniques based on mapping logic or bit flipping, can also be used [13, 16].
- *Test response compression techniques [11, 14, 18]:* The output response can be compressed using combinational compactors such as X-Compact [18] or signature analysis techniques, such as Multiple Input Signature Registers (MISRs) [11,14].

In the next section the proposed compression technique will be described.

3 Compression Technique

Typically, Automatic Test Pattern Generation (ATPG) consists of two steps [29]:

1. In the first step pseudo-random (or optionally weighted pseudo-random) test vectors are applied until the fault coverage exceeds a certain specified fault coverage threshold. Note that this step is skipped in certain ATPG tools or flows [29].
2. In the second step, the remaining undetected faults are targeted using deterministic ATPG (together with static and/or dynamic compaction). The resulting vectors contain both specified bits, also called *care bits*, and unspecified bits, also called *don't care bits (d)*. Typically, the unspecified bits are filled randomly by the ATPG to get additional collateral coverage of un-modeled faults or to get additional coverage of easy detectable faults in case the first step was skipped. Since ATPG targets specific faults in the second phase, a very high percentage of bits are don't care bits. In fact, IBM reported that in their designs about 98% of the bits are don't care bits [19].

The proposed compression techniques will exploit the sparseness of the care bits. The techniques will be explained by introducing the four aspects of the technique: (1) the creation of packets, (2) the grouping of packets, (3) the encoding of the packet groups, and (4) the modified ATPG flow.

3.1 Creation of packets

In [24] a new compression technique is described. A more formal framework will be given here. The technique is based on creating packets. A packet is defined in definition 3.1.1.

Definition 3.1.1: Packet - A packet is a sequence of *0*s, *1*s, and *don't care* bits (*d*). The length of a packet is defined to be the number of bits in the packet.

The *total input test data volume* for a test is defined as the number of used input channels (*e.g.*, the number of DUT scan chain inputs) multiplied by the number of tester cycles that are required for the test, see Fig. 1. Based on the input data volume, the packets can be created *horizontally* by using bits across multiple input channels in the same clock cycle, or *vertically* by using a consecutive sequence of bits fed to the same input channel (see Fig. 1).

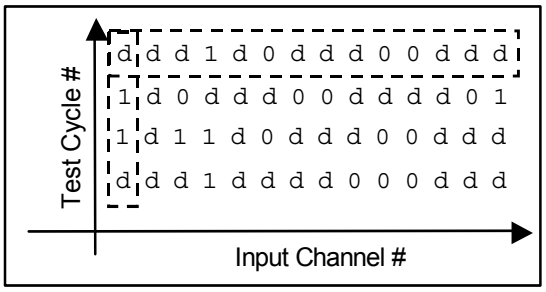

Figure 1:Data Volume: Horizontal and Vertical Packet

A property of each data packet is its match type(s). This property will be defined in definitions 3.1.2 and 3.1.3.

Definition 3.1.2: Compatible Sequences - A sequence of 0s and 1s is compatible with a data packet, if the data packet can be represented by the sequence, i.e. there are no conflicting bits.

Definition 3.1.3: Match Type - The match type is a property of a packet. The match types are defined in Table 1.

Table 1: Different Data Packet Types

Type	Description
Zero Match (L)	Data packet is compatible with a sequence of 0s
One Match (H)	Data packet is compatible with a sequence of 1s
Pseudo-random Match (P)	Data packet is compatible with a pseudo-random sequence, i.e. the care bits do not conflict with the pseudo-random bits.
No Match (N)	Data packet is not of the L, H, or P - type.

Note that since a data packet can be compatible with multiple sequences, a data packet can have multiple types. The data packets of the L, H, and P - type do not have to be stored in the tester memory, but can be

generated by a 0-source (Ground), a 1-source (Vdd), or a pseudo-random source (LFSR).

In case of conventional LFSR reseeding techniques, the LFSR needs to be designed such that all care bits in the pattern can be covered. Note that in our technique the LFSR only needs to supply a pseudo-random source. Hence, a low degree LFSR is sufficient. Note that the degree of the LFSR is unrelated to the packet size.

The following examples will illustrate the match type of data packets. Assume that the used LFSR has the polynomial $f(x) = x^4+x^3+1$ and in the next 4 clock cycles the LFSR generates 0, 1, 0 and 1 subsequently .

Example 3.1.4 **-** Consider the data packet 0ddd. The data packet is compatible with both the 0000 sequence and the LFSR sequence. Hence, the 0ddd data packet is of L/P-type.

Example 3.1.5 **-** Consider the data packet dddd. This packet can be replaced by the 0000, 1111, and the LFSR sequence. Hence, it is of the L, H and P - type.

The original test data volume can be compressed by replacing each original L/H/P-type data packet by a new *compressed data packet* of reduced size without losing information. The compressed data packet consists of bits that designate the type of the data packet (*control bits*), see for example Table 2. If the data packet is of N-type (i.e. the data packet cannot be matched with a 0-source, 1-source, or LFSR source), then the "compressed" data packet includes in addition to the control bits the original data packet.

Table 2: Binary Encoding of the Match Types

Type	Type Control Bits
Zero Match (L)	00
One Match (H)	01
Pseudo Random Match (P)	10
No Match (N)	11

The following example will illustrate the encoding based on matching packets, the binary encoding described in Table 2.

Example 3.1.6 - Consider the following test vector that is separated in packets of 8 bits: "d0dddd0d d01dddddd ddddddd d1dd1ddd ddd1d111 d0dddddd 01ddddd0 ddddddd 10ddddd". Moreover, the used pseudo-random source is a 3 bit LFSR with polynomial $f(x) = x^3+x^2+1$. The LFSR will repeat the 7 bit "1011100" sequence. Hence, the data packets are of respectively L/P-type, P- type, L/H/P-type, P/H-type, P/H-type, P/L-type, P-type, L/H/P-type, and N-type. Consequently, the test vector can for example be compressed in the following sequence (using the control bit encoding from Table 2): "00.10.01.01.10.00.10.01.11.10101111".
Note that this sequence is only 1 sequence out of the 144 possible compressed sequences (2•1•3•2•2•2•1•3). In this example, the original 9•8=72 bits are compressed to 26 bits, of which 18 bits are control bits, i.e. 69% of the compressed bits are control bits.

In the previous example, it became clear that the control bits could become a significant percentage of the compressed test data volume. The next sections describe how packets can be combined and how binary encoding techniques can be used to significantly reduce the control bit percentage.

3.2 Grouping of packets

Definition 3.2.1, 3.2.2, and 3.2.3 will define the match packet group, match packet size, and match packet type.

Definition 3.2.1: Match Packet Group - A match packet group is defined as a number of consecutive data packets that have a common type. For example, a data packet of L/P–type can be grouped with a data packet of P–type, because they have the P-type in common.

Definition 3.2.2: Match Group Size - The group size is defined as the number of data packets in a group.

Definition 3.2.3: Match Group Type - The group match type is defined as the common match type(s) of the packets included in the group. Hence, the type of the group can be zero (L), one (H), pseudo random (P), or no match (N).

Instead of storing control bits to encode the type of each packet of a group, only the type of the group needs to be encoded. Hence, for every packet that can be included in a group, the packet type control bits do not have to be stored. However, note that now the group size needs to be encoded and stored.

Finding the optimal mapping of which packets should be included in which groups, depends on many factors. The mapping problem can be analyzed as a covering problem, in which all packets need to be covered by the minimum number of maximum sized groups. In that case, the number of control bits is minimized. To simplify the problem of mapping packets to groups, we assume that fixed-length codes are used, i.e. the number of control bits is independent of the group size and group type. Moreover, we assume that there are no restrictions on the number packets that can be combined in a group. Under these circumstances, the following greedy algorithm applies.

Grouping Algorithm

1. Get the next compressed data packet type
2. Select type that maximizes group size
3. Merge all data packets of selected type into one group
4. If there are remaining packets,go to 1.

The following examples will illustrate the grouping algorithm.

Example 3.2.4 **-** The data packets of type P/L, P/L, L, L, N, L, P/L, L, H/P, H/P, H/P can be compressed in 4 match groups:L(size 4), N(size 1), L(size 3), P/H(size 3).

Example 3.2.5 **-** The test vector in example 3.1.6, i.e. the packets of type L/P, P, L/H/P, P/H, P/H, P/L, P, L/H/P, and N, can be compressed in 2 match groups: a P - type group of size 8 and a N - type group of size 1. If we assume that the group sizes are encoded using a normal 4 bits fixed-length binary code, then the original test vector can be compressed to "10.1000.11.0001.10101111", reducing the number of control bits from 16 to 10.

3.3 Encoding of the packet groups

In previous examples, the group sizes and group types were encoded using fixed-length code words (e.g. see Table 2). This section will describe the disadvantages of fixed-length code words and will propose alternatives.

3.3.1 Fixed-length encoding

If there are no restrictions on the number of packets in a group and all possible group sizes are encoded with their own fixed-length code word, then the required number of control bits for each codeword equals $m=log_2(groupsize_{max})$. The *overhead* is defined as the ratio between the control bits and the data bits. Even though the m control bits overhead is acceptable for the maximum sized group (*e.g.* a 10 bit codeword for a group size of 1024 packets is acceptable), for the majority of smaller group sizes the m control bits overhead may be unacceptable (*e.g.* a 10 bit codeword for a group size of 2 packets may be unacceptable).

To solve this problem, only a limited number of group sizes are represented by an own codeword. In other words, only a limited number of group sizes are *directly encoded*. For example, instead of encoding all group sizes, only the 8 group sizes described in Table 3 are directly encoded. In that case, a group size of 1024 packets can be created by splitting the group into 8 smaller directly encoded groups (*subgroups*) of a group size of 128 packets. Moreover, a group of size 2 can be directly created using only 3 control bits (001).

The set of the minimum number of (directly encoded) subgroups for a given required group size, can be found by using a simple greedy algorithm. The first subgroup is the largest directly encoded group, with a group size smaller than the required size. The remaining gap between the required size and the already created size is iteratively filled with the next largest directly encoded group size that fits the gap.

The optimal set of available directly encoded group sizes depend on the probability of the occurrence of certain group sizes (i.e. group sizes that occur rarely don't need to be encoded directly with own code words).

3.3.2 Variable-length encoding

Instead of fixed-length code words, Huffman codes can be used to attach shorter code words to group sizes and/or group types that occur with a higher probability [15]. The codes can be derived using a Huffman tree. By again only directly encoding a limited number of group sizes, the hardware implementation can be kept simple. Table 3 gives an example of 8 Huffman codes.

Table 3: Binary encoding techniques for the group size

Size	Prop.	Fixed-length	Huff-man	Golomb-Rice	
				Prefix	Tail
1	60%	000	01	1	001
2	20%	001	100	1	010
3	10%	010	001	1	011
4	4%	011	110	1	100
8	2%	100	101	01	000
16	2%	101	111	001	000
32	1%	110	0000	$00001=0^4 1$	000
128	1%	111	0001	$0^{16}1$	000

By using a Golomb-Rice encoding [3,12], all group sizes can be directly encoded, while still maintaining a simple hardware decoder implementation. A Golomb-Rice codeword consists of a concatenation of a variable-length prefix of n bits and a fixed-length tail of m bits (representing a decimal value of q), making the total length of the codeword $n+m$ bits. The n bit prefix starts with $n-1$ 0s and the last bit is always 1 (notation: $0^{n-1}1$). This way, decoding the prefix (n) is reduced to only counting the number of 0s. The tail is a normal binary fixed-length code. The value of a Golomb-Rice codeword is defined as $n \bullet 2^m + q$

For clarity, Fig. 2 shows the different steps in the compression approach. Row 1 shows the uncompressed data stream, separated in packets of 4 bits. Row 2 shows the LFSR sequence based on the $f(x) = x^4+x^3+1$ polynomial. Note that the LFSR sequence repeats every 15 bits, whereas the packet size is only 4 bits. Row 3 shows the different packet types, based on comparing the data stream packets to the LFSR sequence. Row 4 shows the result of combining packets in groups (the group size is designated between parentheses). Row 5 shows the result of creating the required group sizes using subgroups of directly encoded sizes (see Table 2). Finally, row 6 shows the resulting compressed data stream (based on the Huffman size encoding described in Table 3 and the fixed-length type encoding described in Table 2).

3.4 ATPG flow

If ATPG is performed without the initial pseudo-random test vector phase, then not filling the *don't care* bits of patterns with random values will increase the number of patterns significantly. This is because the potential extra fault coverage of the random filled patterns is not exploited.

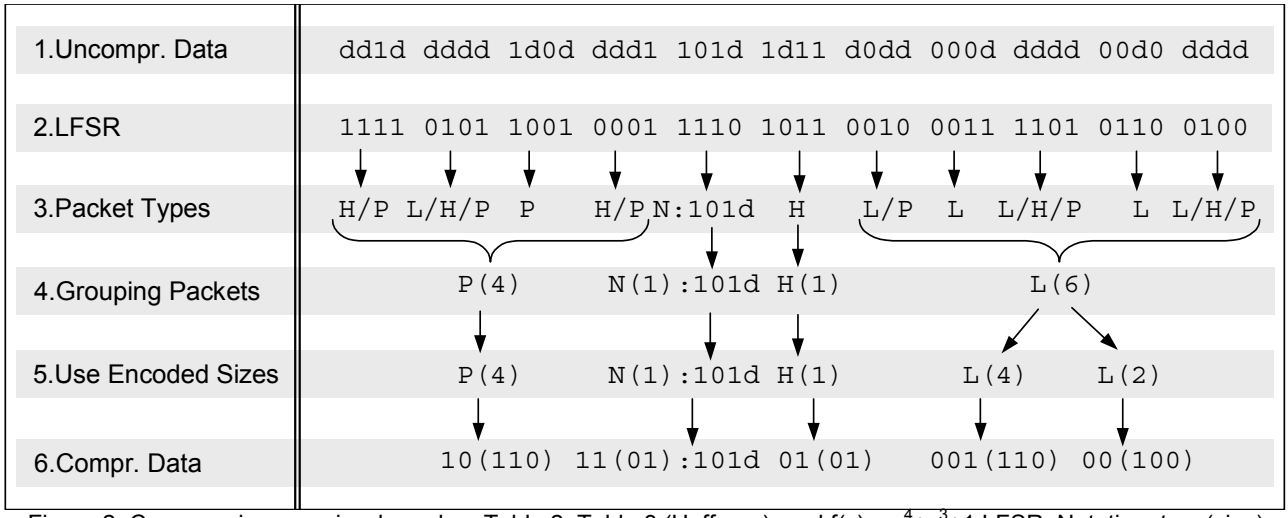

Figure 2: Compression overview based on Table 2, Table 3 (Huffman), and $f(x) = x^4+x^3+1$ LFSR. Notation: type(size)

There are 2 possible ATPG flows that prevent an increase in the number of patterns:

1. It is possible to match the fill sequence during ATPG with the pseudo-random fill of the decompression hardware. In this case, the number of patterns will not increase. This approach however, requires modifying the ATPG tool. This is not possible in case a commercial ATPG tool is used.
2. A solution that does not require modifying the ATPG tool is described in the algorithm below.

ATPG Flow Algorithm

1. Drop easy detectable faults from fault list (by using random patterns until 60% coverage
2. Do ATPG without random fill for remaining faults in the fault list
3. Perform compaction (if needed extra compaction compared to the normal flow)
4. Perform the LFSR based pseudo-random fill (*e.g.*, by using a system call in case of a commercial tool)
5. Fault grade the filled vector set
6. Perform deterministic ATPG, in case a few easy detectable faults are not covered by the random fill

If in this flow the number of no random fill patterns still increase compared to the number of random fill patterns, then step 2, 3 and 4 can be repeated multiple times on a smaller number of patterns.

Note that in both flows the compression algorithm can be applied on the fully compacted ATPG set and that there are no restrictions in the number of care bits per pattern.

The next section will explain the experimental results of applying the compression techniques on industrial devices.

4 Experimental Results

The compression techniques were applied on two large industrial designs implementing network processors with the characteristics, described in Table 4.

Table 4: Design Characteristics of the ASICs

Parameters	ASIC 1	ASIC 2
Design Size	~7M gates	~300k gates
Total Number of F/F	~105k	~30k
Total Number of Scan Chains	23	10
Total number of Collapsed Faults	~6M	~300k
Test Data Volume	~1.6G	~18M

The test vectors for the considered designs were generated using a commercial ATPG tool, using flow 2 described in section 3.4.

By initially dropping the easy detectable faults (step 1) and by extra compaction (step 3), an increase in the number of vectors without random fill could be prevented. Only one design (ASIC1) resulted in a 1.5x increase in the number of vectors compared to the normal ATPG set. This was due to memory limitation problems during the extra compaction (step 3). Note that if flow 1 (section 3.4) were used, then there would have been no increase in the number of patterns. After automatic test pattern generation, the generated vectors were written in the STIL format.

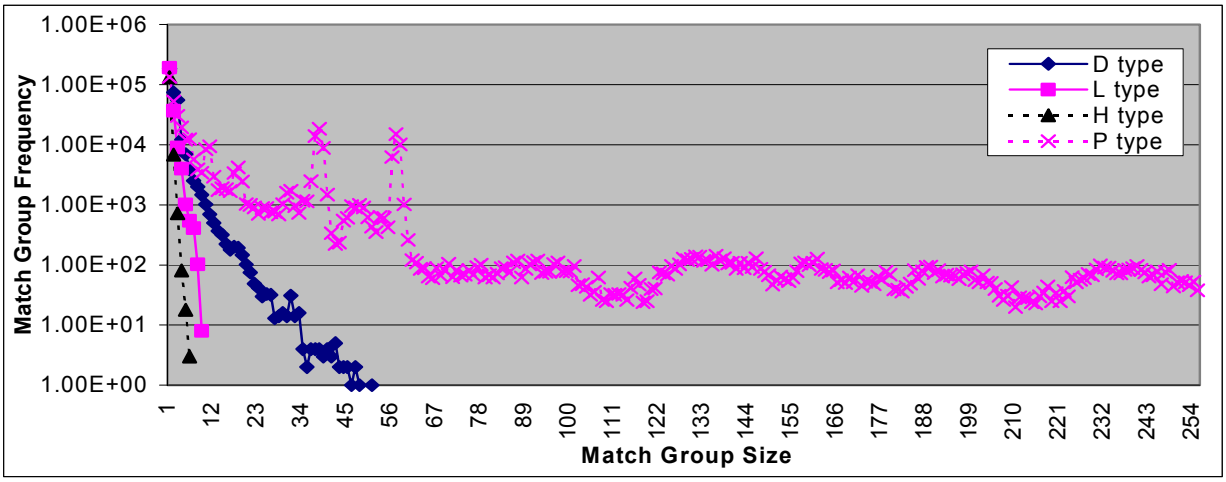

Figure 4: The group frequency as function of the specific match group sizes, for the 4 different match types (ASIC1)

After STIL generation, a PERL script was used to extract the scan input vectors from the STIL output file. A software package has been developed to perform the described compression techniques on the extracted scan input vectors. The compression software was executed on a HP 9000/785/J5600 machine with 2 550MHz PA-RISC processors and 2G of physical RAM.

The experimental results will focus on showing the compression ratio. For completeness, this ratio is defined in equation 1.

$$\text{CompressionRatio} = \frac{\text{Total bits without compression}}{\text{Total compressed bits}} \qquad [1]$$

4.1 Creating packets and encode packets

The graph in Fig. 3 shows the results of applying the compression techniques with fixed-length type encoding and without grouping (see Table 2) on ASIC 1 as function of the packet size. The results are shown for different types of compression:

- **P/N** designates the compression ratio if only P and N types are used for compression.
- **L/H/N** designates the compression ratio if only L, H, and N types are used for compression.
- **L/H/P/N** designates the compression ratio if all types are used for compression.

The figure shows that for this design a maximum compression ratio of 16.8x can be achieved by using L/H/P/N-type compression and a packet size of 64 (also see [24]). The total compression run time for all these scenarios was about 4.5 hours.

The figure also shows the dependency of the compression ratio on the packet size. The compression ratio is smaller in case of small packet sizes, because the ratio of the control bits to the data bits for the packet, i.e. the overhead, is larger. Increasing the packet size increases the compression ratio as control bits replace a higher number of data bits. However, increasing the packet size beyond a certain point will reduce the probability of finding L/H/P-type packets. Increasing the packet size further, will reduce the probability of matching a packet, effectively reducing the compression ratio.

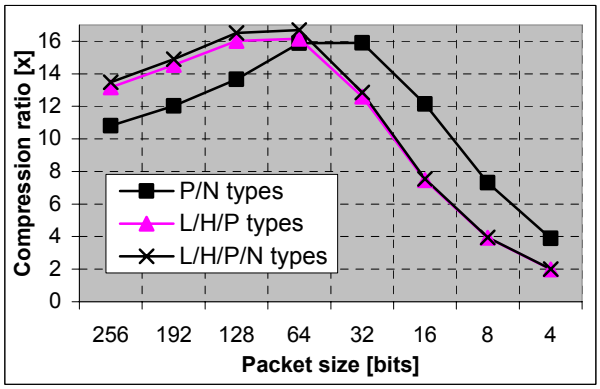

Figure 3: Compression results on ASIC1 without grouping and with fixed-length type codes and Huffman size codes

4.2 Group packets and encode groups

The graph in Fig. 4 gives the *group frequencies* (i.e. the number of occurrences) for the N, L, H, and P-type groups versus the different group sizes, in case the grouping algorithm is applied on ASIC 1. The figure shows that the maximum group size of the H and the L-type groups are respectively 5 and 10 packets. The maximum group size of the N-type groups is 55 packets. Hence, the N-type compression due to grouping will be bigger than the H and L-type compression, because larger groups can be created (i.e. making the control bit overhead smaller). Moreover, the biggest compression ratio due to grouping will occur by introducing the P-type groups.

Table 5: Experimental results of applying the compression technique on 2 industrial devices

Circuit Name	Input Test Data Volume	Compressed Test Data Volume									
		No Grouping		Grouping with fixed size encoding		Grouping with group type Huffman encoded		Grouping with group type and size Huffman Encoded		Grouping with group size Golomb-Rice Encoded	
		Packet Size	Total # Bits	Packet Size	Total # Bits	Packet Size	Total # Bits	Packe tSize	Total # Bits	# Tail bits in code word(*m*)	Total # Bits
ASIC 1	1,600M	16	213M	16	42M	16	41M	16	37M	6	70M
		32	125M	32	**33M**	32	**32M**	32	**29M**	7	61M
		64	**96M**	64	54M	64	53M	64	52M	8	**52M**
		128	97M	128	77M	128	77M	128	76M	9	60M
ASIC 2	18,249k	16	2,946k	4	13,705k	4	10,511k	4	13M	4	2,370k
		32	2,152k	8	**1,124k**	8	**1,033k**	8	**1,078k**	5	2,001k
		64	**2,087k**	12	1,307k	16	2,030k	12	1,274k	6	**1,911k**
		128	2,574k	16	1,753k	32	1,287k	16	1,723k	7	1,958k

* Based on statistical analysis the {1,2,3,4,8,16,32,128} packet sizes are selected (in case of Huffman: see codes in Table 3).

Fig. 5 gives the actual L, H, and P-type compression ratios of encoding the groups using different encoding schemes. Since these different scenarios did not have a significantly different impact on the N-type compressed data volume, this data volume is not shown.

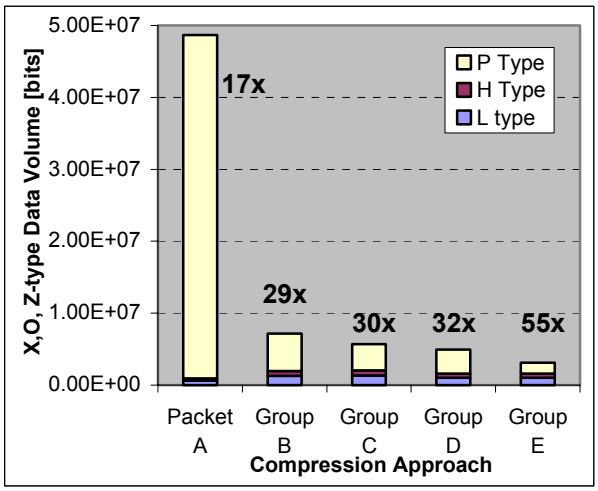

Figure 5: P, H, and L-type compressed data volume and the corresponding compression ratios for approach A,B,C,D and E.

The following group encoding schemes are considered:

- The results of encoding each 64-bit packet individually, without using the grouping algorithm, are represented by the *packet A* bar. The types are encoded using the fixed-length code words, see Table 2. The compression ratio equals 16.8x.
- The results of applying the grouping algorithm on 64 bit packets, and encoding the groups using 8 different group sizes ({1,2,4,8,16,32,64,128}), with fixed-length code words, is represented by the *group B* bar. The compression ratio equals 28.8x.
- The results of applying the grouping algorithm on 64 bit packets, and encoding the groups using 10 different Huffman encoded group sizes ({1,2,4,8,16,32,64,128,256,512}), are represented by the *group C* bar. The compression ratio equals 30.1x.
- The results of applying the grouping algorithm on 64 bit packets, and encoding the groups using 9 different Huffman encoded group sizes ({1,2,4,8,16,32,64,128,256}), are represented by the *group D* bar. The compression ratio equals 31.6x. This is better than the previous scenarios, because the 9 different codes enable a slight improvement in L/H-type volume compression.
- The results of applying the grouping algorithm on 32 bit packets, and encoding the groups using the 8 different Huffman encoded group sizes from Table 3 ({1,2,3,4,8,16,32,128}), are represented by the *group E* bar. The compression ratio equals 55x.

Table 5 gives experimental results of applying the compression techniques using different packet sizes. In addition, it shows the results of applying the compression techniques on the second industrial design (ASIC2). Note that the additional advantage of applying a Huffman encoding compared to the fixed-length encoding is very limited. This is due to the limited number of directly encoded sizes.

5 Implementation Examples

This section will describe different implementation examples. The described techniques can be implemented (1) using ATE and on-chip logic, (2) using only on-chip logic, and (3) using only ATE.

5.1 Using ATE and on-chip logic

Fig. 6 shows a decompression unit architecture that can be used in case the code words are of a fixed bit length.

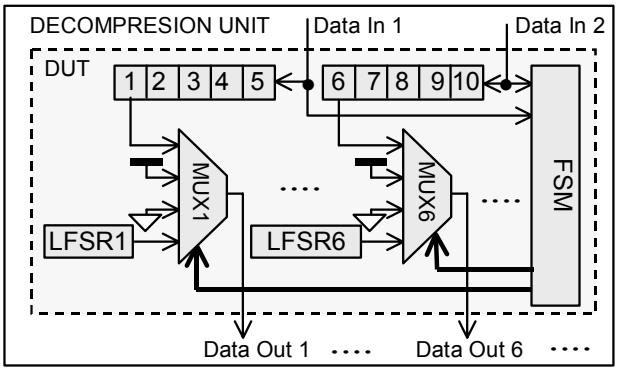

Figure 6: Decompression Unit

The architecture consists of a Finite State Machine (FSM) and an internal shift register consisting of two 5-bit shift registers connected to the 2 input channels.

Moreover, for each of the 10 output channels, a multiplexer and a data generator (i.e. a 0, 1, and LFSR source) is required.

In the architecture, the number of input channels of the decompression unit is equal to the number of bits of the type codeword (i.e. 2 bits). A packet is defined horizontally as the required output bits at a certain tester cycle, see Fig. 1. Hence, the number of output channels is chosen equal to the optimal packet size. In this example, an optimal packet size of 10 bits is assumed (in case of the experiments in Fig. 3, a packet size of 64 was used).

The FSM decodes the packet type, designated by the 2 input channels:

- In case of a N-type (no match) packet, the FSM directs the internal shift registers to serially shift in the data packet in 5 tester cycles using the two input channels (5*2 bits). After the register is filled with the 10 bits of the packet, the 10 bits of the packet can be directed to the 10 output channels in one tester cycle (10*1 bits). Since the number of output bits equals the number of input bits, the input bandwidth equals the output bandwidth.
- In case of a P/H/L-type packet, the FSM sets all multiplexers to direct the appropriate internal data generator (pseudo-random, 0, or 1) to the 10 outputs in one tester cycle. In other words, without the decompression unit we would need 5 tester cycles to feed in the 10 bits through the 2 channels, and with the decompression unit we only need 1 cycle. Hence, a 5x test time reduction for P/H/L-type packets is achieved in this example.

More details about the scheme can be found in [24]. When multiple packets are grouped, the same type codeword needs to be repeated multiple times. This can be achieved by using a ATE repeat-count instruction.

This ATE instruction enables repeating a certain bit value a number of tester cycles. Note that depending on the ATE architecture, the instruction op-code overhead does not count towards the critical data memory requirements, i.e. it counts towards the non-critical instruction memory requirements.

5.2 Using only on-chip logic

An additional test time reduction can be achieved by implementing the repeat-count on-chip, instead of relying on the ATE. When the decompression unit is busy decoding a group, the input channels can be multiplexed towards another decompression unit. Techniques to enable this are typical for compression techniques using compression code words. In [3] a good overview of the techniques, tradeoffs, and benefits, is given.

The additional hardware of a variable-length type/size decoder is small, because the number of directly encoded sizes is very limited (*e.g.*, only 8 directly encoded sizes). The experiments showed that variable-length codes do not result in a significant increase in the compression ratio, so we will not describe the implementation in much detail. A low overhead Huffman decoder implementation can be found in [31], and a low overhead Golomb-Rice decoder implementation can be found in [3].

To design an optimal decoder, the optimal set of directly encoded group sizes, and the corresponding group probabilities, need to be calculated. This can only be done when the ATPG vector set is available. Typically, the ATPG vector set is only available after the design is ready. Hence, if the design is modified such that the original optimal decoding is not optimal anymore, then the decoder hardware might need to be modified.

5.3 Using only ATE

If the main objective is reducing ATE vector memory requirements and reducing test time is not the main concern, then the decompression techniques can be implemented on the ATE. The advantage of an ATE implementation is that no on-chip decompression hardware is required.

The implementation is similar to the architecture of Fig. 6, with only one input channel: the FSM detects the type and size string serially. Moreover, unless the scheme is used for multiplexing ATE channels, the decompression unit has only one output. (and packets are created vertically instead of horizontally, see Fig. 1).

Circuit Name	Number of Scan Flip Flops	Mintest [28]		Illinois Scan Architecture [20]		FDR Codes [6]		Linear Decompressors [30]		Proposed Approach	
		Number of Vectors	Number of Bits	Number of Vectors	Number of Bits	Number of Vectors	Number of Bits	Number of Vectors	Number of Bits	Number of Vectors	Number of Bits
S13207	700	233	163k	273	110k	236	31k	251	24k	250	7k
S15850	611	96	59k	178	33k	126	26k	170	16k	257	5k
S38417	1664	68	113k	337	96k	99	93k	296	94k	358	8k
S39694	1464	110	161k	239	96k	136	78k	182	35k	137	8k

Table 6: Comparison of the compression techniques with other compression techniques

6 Comparison with Other Approaches

6.1 Compression using compression codes

The proposed compression techniques fit best in the category of compression using compression codes.

When determining the packet size, all possible sizes are considered, including a packet size of 1. The scenario in which the packet size is 1 can be considered a pure 3-symbol run-length encoding. However, due to the overhead of encoding the type and the fact that only a limited number of group sizes are encoded directly, our technique of introducing packets, i.e. a packet size larger than 1, turns out to be more effective.

Maintaining the don't care information during compression, enables us to get a high compression ratio, while still maintaining pseudo-randomness where it is needed. Other 2-symbol compression techniques that don't exploit don't care bits, result in a significantly lower compression ratio.

To increase the compression ratio of 2-symbol compression techniques, the don't care bits can be filled with constant 0s/1s or the last care bit can be repeated [3,6,26]. However, due to the reduced coverage per patterns the number of patterns will increase. In case of an on-chip implementation of the decompression unit, the effect of fanning out to multiple scan chains might compensate the effect of the increase in the number of patterns. In case of an ATE implementation, an increase in the number of decompressed patterns is often simply unacceptable.

Whereas the proposed techniques have the described advantages, the on-chip hardware overhead is comparable to other compression techniques based on using compression codes [3]. Previously published techniques used ATE repeat-count instructions for the purpose of compression [26]. However, a repeat-count of random bits was not possible. One of the implementation examples of the proposed technique enables ATE repeat-count instructions for random fill, by combining it with on-chip logic.

6.2 Compression based on LFSR reseeding

The technique does not require the calculation of LFSR seeds. In addition, the LFSR can be of a very low degree, i.e. only a few flip-flops. Moreover, the LFSR doesn't have to be (re)set each pattern. In addition, there are no restrictions to the maximum number of care bits per pattern. The compression technique can be applied on the fully dynamically and statically compacted set. In addition, a higher probability of a certain bit value within a certain consecutive sequence, also called *clustering*, can be exploited (e.g. consecutive padding bits due to a mismatch in scan-chain lengths).

In addition, the proposed technique is able to create consecutive sequences of ones and zeros easily, for example to prevent bus contention, illegal states, or to reduce scan power dissipation.

6.3 Pseudo-random BIST techniques

Typically, the compression techniques based on pseudo-random BIST techniques, require more decompressed patterns. This will result in the same power dissipation and ATE implementation issues, described before.

All proposed techniques are also applied on ISCAS'89 benchmark circuits. Table 6 describes the best compression results compared with other published techniques [27].

7 Conclusions

The problems of having a limited ATE vector memory, having long upload times, and having limited I/O bandwidth, are among the most critical problems the industry is facing.

This paper explored several compression techniques based on creating packets, grouping packets, and encoding packets using both fixed-length encoding techniques and variable-length encoding techniques. Compared to previous work, the techniques have several advantages.

The proposed compression techniques are applied on multiple industrial designs and benchmark circuits,

resulting in compression ratios varying from 17x-70x, on top of static and dynamic compaction techniques.

The compression algorithms are simple to implement and the impact on the overall design and test flow is limited. There are no hard tool requirements on the proposed compression techniques. It only requires an ATPG tool, which can generate *don't care* bit information.

8 Acknowledgments

The authors would like to thank Professor McCluskey, James Li and Chao-Wen Tseng of the Center for Reliable Computing at Stanford University for their valuable discussions on the paper and the experimental results. This work was sponsored by Agilent Technologies.

9 References

[1] International Technology Roadmap for Semiconductors (ITRS), 1999.

[2] G. Hetheringten, T. Fryars, N. Tamarapalli, M. Kassab, A. Hassan and J. Rajski, "LBIST for Large Industrial Designs", Proc. Of International Test Conference", pp. 358-367, 1999.

[3] A. Chandra, and K. Chakravarty, "Test Data Compression and Decompression for System-on-a-chip using Golomb codes", VLSI Test Symposium, pp. 113-120, 2000.

[4] A. Khoche and J. Rivoir, "I/O Bandwidth Bottleneck for Test:Is it Real?" IEEE Test Resource Partitioning Workshop, pp. 2.3-1 - 2.3-6, 2000.

[5] M. Ishida, D.S Ha and T. Yamaguchi, "COMPACT: A Hybrid Method for Compressing Test Data", VLSI Test Symposium, pp. 62-69, 1998.

[6] A. Chandra and K. Chakravarty, "Frequency Directed Run-length Codes with applications to System-on-a-chip", VLSI Test Symposium, pp. 42-27, 2001.

[7] B. Koenemann, "LFSR-Coded Test Patterns for scan Designs", Proceedings of European Test Conference", pp. 237-242, 1991.

[8] S. Hellerbrand, J. Rajski, S Tarnick, S. Venkatraman and B. Courtois, "Built-in Test for Circuits with Scan Based Reseeding of Multiple Polynomial Linear Feedback Shift Registers", IEEE Transaction on Computers, vol. 44, No. 2, pp.223-233, 1995.

[9] A. Jas, J. Ghosh Dastidar and N. A. Touba, "Scan Vector Compression Using Statistical Coding", VLSI Test Symposium, pp. 114-120, 1999.

[10] I. Bayraktaroglu and A. Orailoglu, "Test Volume and Application Time Reduction Through Scan Chain Concealment", Design Automation Conference, pp. 151-155, 2001.

[11] P. H. Bardell, W. H. McAnney and J. Savir, "*Built-in Test For VLSI: Pseudo-random Techniques*", Wiley Inter-Science, 1987.

[12] S.W. Golomb, "Run-length encoding," IEEE Trans. Inform. Theory, vol.. 11-12, pp.399-401, Dec. 1966.

[13] G. Kiefer and H-J. Wunderlich, "Application of Deterministic Logic BIST on Industrial Circuits", International Test Conference (ITC). pp. 105-114, 2000

[14] K. Barnhart, B. Keller, B. Konenmann and R. Walther, "OPMISR: Accelerated scan with On product signatures", IEEEE European Test Workshop (ETW), 2001.

[15] D.A. Huffman, "A Method for the Construction of Minimum Redundancy Codes", Proc. of IRE, Vol.40, No.9, pp. 1098-1101, 1952.

[16] N.A. Touba and E.J. McCluskey, "Altering a Pseudo-Random Sequence of Bits for Scan-Based BIST", Proc. of IEEE International Test Conference (ITC), pp. 167-175, 1996.

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

[18] S. Mitra and K. S. Kim, "X-compact: An Efficient Response Compaction for Test Cost Reduction", To appear in Proc. of the IEEE International Test Conference (ITC), 2002.

[19] B. Koenemann, C. Barnhart, B. Keller, T. Snethen, O. Farnsworth, D. Wheater, "A SmartBIST Variant with Guaranteed Encoding", IEEE Test Resource Partitioning Workshop, pp.2.4.1-6, 2001

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

[21] F. F. Hsu, K. M. Butler, J. H. Patel, "A Case Study on the Implementation of the Illinois Scan Architecture," Proc. of the IEEE International Test Conference (ITC), pp. 538-547, 2001.

[22] E. J. McCluskey and C.W. Tseng, "Stuck-Fault Tests vs. Actual Defects," Proc. of the International Test Conference, 2000.

[23] TestKompress datasheet, http://www.mentor.com/dft/testkompress/test_kompress_ds.pdf

[24] A. Khoche, E.Volkerink, J. Rivoir and S. Mitra, "Vector Compression using EDA/ATE Synergies", IEEE VLSI Test Symposium, pp. 97-102, 2002.

[25] W.B. Pennebakerand J. L. Mitchell, "*JPEG Still Image Data Compression Standard*", Van Nostrand Reinhold, 1993.

[26] K. Barnhart, B. Keller, B. Konenmann and R. Walther, "OPMISR: The Foundation for Compressed ATPG Vectors", ITC 2001.

[27] C.V. Krishna and N. A. Touba, "Reducing Test Data Volume Using LFSR Reseeding with Seed Compression", To appear in Proc. of the IEEE International Test Conference (ITC), 2002.

[28] I. Hamzaoglu and J. Patel, "Test Set Compaction Algorithms for Combinational Circuits" Proc. of International Conference on Computer-Aided Design (ICCAD), pp. 283-289, 1998.

[29] Tetramax Userguide, Synopsys, 1998.

[30] I. Bayraktaroglu, and A. Ogailoglu, "Test Volume and Application Time Reduction Through Scan Chain Concealment," IEEE Proc. of Design Automation Conference, pp. 151-155, 2001.

[31] A. Mukherjee, N. Ranganathan, and M. Bassiouni, "Efficent VLSI Designs for Data Transformation of Tree-based Codes," IEEE Transactions on Circuits and Systems, pp. 306-314, March 1991.