BITSTREAM COMPRESSION TECHNIQUES FOR VIRTEX 4 FPGAS

Radu Ştefan [*], *Sorin D. Coţofană*

Computer Engineering Laboratory,
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands
email: R.A.Stefan@tudelft.nl, S.D.Cotofana@ewi.tudelft.nl

ABSTRACT

This paper examines the opportunity of using compression for accelerating the (re)configuration of FPGA devices, focusing on the choice of compression algorithms, and their hardware implementation cost. As our purpose is the acceleration of the configuration process, estimating the decoder speed also plays a major role in our study. We evaluate a wide range of well-established compression algorithms and we also propose two methods specifically developed for compressing FPGA configuration bitstreams, one based on a static dictionary and the other on arithmetic coding. For the arithmetic coding we propose a statistical model that takes advantage of the particularities of the configuration bitstreams of the Virtex 4 FPGA family. We evaluate the efficiency of the proposed methods along with state of the art compression algorithms on a number of benchmark circuits, some selected from the available open source implementations and some synthetically generated. Our evaluations indicate that using modest resources we can achieve parity and even exceed comercial software in terms of compression ratio, and outperform all other traditional algorithms. All our implemented decompressors are shown to use less than 1.5% of the slices available on the FPGA device.

1. INTRODUCTION

Field Programmable Gate Arrays (FPGAs) are, as their name suggests, arrays of configurable blocks, connected by configurable routes. As the number of blocks and the complexity of the routing resources have increased so has the amount of memory needed to store the configuration data and the time needed to upload these data on the chip. Bitstream compression has been identified by previous studies [1, 2, 3] as a possible solution for reducing bitstream storage requirements and accelerating FPGA (re)configuration.

One of the major FPGA manufacturers, Altera has already decided to incorporate decompression hardware into their products as it is the case with Stratix II family of FPGAs [4]. In this case, the compression method has been chosen by the manufacturer and it is hardwired into the product. Decompression has also been supported in the past by external configuration devices, such as the EPC [5] and System ACE MPM [6]. However, the FPGAs in the Virtex 4 family [7] are capable of decompressing the bitstream internally, with a decompressor implemented on the actual FPGA fabric, as indicated in studies of Huebner [8]. This scheme is generic, but the question of which algorithm is more appropriate for the task of bitstream compression remains open.

In this line of reasoning, we make the target of this study the evaluation of a wide range of compression algorithms. In addition to the well-established compression methods we propose two novel techniques, one based on arithmetic coding and the other using a fixed dictionary.

To evaluate the implications of our proposal we considered a number of benchmark circuits, mapped them to a Xilinx Virtex 4 FPGA and compressed the obtained bitstreams with our methods as well as with other state of the art compression methods. To give a relative estimation of the compression ratio achieved by our algorithm, we include in our tests a popular and highly effective commercial compression software, RAR [www.rarsoft.com]. The experiments suggest that the first of our methods outperforms all the other methods in terms of compression efficiency. While our method outperforms RAR by a small margin, it is important that it does so while using orders of magnitude less resources. RAR uses a dictionary and data structures in the order of megabytes, while the memory requirements of our algorithm are in the order of tens or hundreds of bytes.

The second proposed method focuses on simplifying the decompression hardware, and thus achieves the highest decompression speed by a margin of 266%, although providing lower compression ratios. We have implemented in hardware decoders for all the algorithms for which we considered an implementation to be feasible and we present their cost and performance for comparison.

The remainder of the paper is organized as follows: Sec-

[*] A part of this research was performed by Radu Ştefan while being affiliated with the Transilvania University of Braşov.

tion 2 presents related studies concerning bitstream compression techniques. In Section 3 we introduce two methods we specifically adapted for the task of FPGA bitstream compression. Section 4 presents experimental results and a comparison between our algorithms and the algorithms described by previous papers. Finally, Section 5 draws some conclusions from these experiments.

2. RELATED WORK

A number of studies have previously targeted FPGA bitstream compression. A notable one [9] focuses on an earlier family of Virtex products. It examines a wide range of techniques, studies stream regularity, the effect of symbol length, frame reordering and readback, a wildcard technique inherited from a previous family of FPGAs, and proposes methods such as Huffman coding and dictionary based compression (LZSS). Arithmetic coding is mentioned, but without a reference to the statistical model assumed. An older paper from the same authors [1] targeting xc6200 FPGAs, exploits a feature of the configuration hardware on that platform, called wildcard registers, that allows programming a selection of multiple rows and columns at once. A subsequent paper [10] addresses the use of runlength encoding for the same family of FPGAs. The use of don't cares has also been proposed in [2] to enhance compression. The method however requires knowledge of the internal structure of the FPGA and the bitstream format, which is not published by the manufacturer in the case of newer devices.

A major direction of research has been the exploitation of inter-frame regularity either by using a previously uploaded frame as a dictionary in a dictionary-based compression method [9] or by computing the XOR difference between frames [3]. Frame reordering is particularly useful for this technique, and complex algorithms like those described in [11] were proposed for this task. Although we have performed experiments in this direction, this method did not provide better results and for reasons of brevity will not be presented here.

A modified LZW dictionary-based compression method, unfortunately having high memory requirements, is presented in [12], while the more simple LZ77/LZSS algorithm is the method of choice in [8].

More recent studies [13] propose using static xor masks dependent on the type of resources found in the FPGA: LUTs, global routing, local routing. Their study targets the same early family of Virtex FPGAs.

In general, the cost of decompression hardware was not addressed in the literature, a notable exception being the study in [8] (LZSS compression).

In our study, we attempt to move the focus onto more recent Virtex FPGAs, that is the Virtex–4 (the Virtex–5 was not available at the beginning of our study). We find that the change in architecture had a major impact on the structure of the configuration bitstreams, as so probably had the evolution of synthesis software. We reproduce for this architecture the most significant experiments described in the literature.

3. PROPOSED ALGORITHMS

In this section we present our two proposed compression methods. Arithmetic coding is in general perceived as an expensive compression method, particularly because of the required multiplication. We show however that a low-cost arithmetic decoder can be obtained with little loss in terms of compression ratio. A second method is designed based on standard dictionary compression methods and focuses on simplicity and speed.

3.1. Bitstream Compression Based on Arithmetic Coding

Arithmetic coding is a technique that allows storing symbols using a fractional number of bits based on the probability of occurrence. Although at first this may seem non-intuitive or even impossible, the actual implementation is rather simple. A detailed description of the algorithm can be found in [14].

Consider a memory unit which holds n bits of data. This unit, say a register, can store 2^n different values. By analogy a hypothetical storage unit, capable of memorizing a value between 0 and $n-1$ would be said to have a capacity of $\log_2 n$ bits, which may be a fractional number. As it turns out such a storage unit is not only possible but it is easy to implement and consists of two registers, one holding the size of the interval [0,n) and the other the actual value.

Information is added to this conceptual storage unit an integer number of bits at a time, by doubling (or multiplying by 2^n) the size of the interval, and choosing a new stored value from a set of 2^n elements, based on the n bits of information added. At each step of the algorithm we add as much information as possible to this storage unit, in order to fully utilize the available register width. This operation is called renormalization. Information is removed from the storage unit by splitting the set of possible values into subsets associated with each symbol to be encoded. For simplicity, the subsets are two disjoint intervals. The elements of the subsets can be seen either as fractions or integer numbers. Ideally, when decoding a symbol and performing a division of the set, the size of each subset should be proportional to the probability of the symbol it represents. Because the number of elements in the set is constrained by the size of the registers, a certain approximation must be tolerated.

Figure 1 presents the structure of the arithmetic decoder. The high cost of the decoder arises from the use of a multiplier circuit. We attempt to reduce the cost, by decreasing

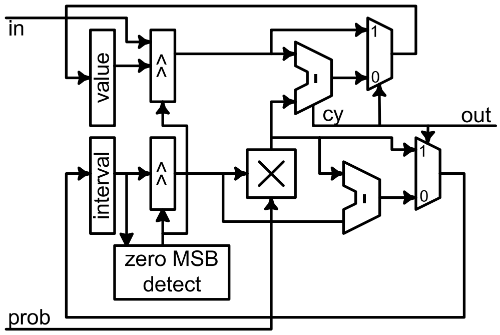

Fig. 1. The arithmetic decoder

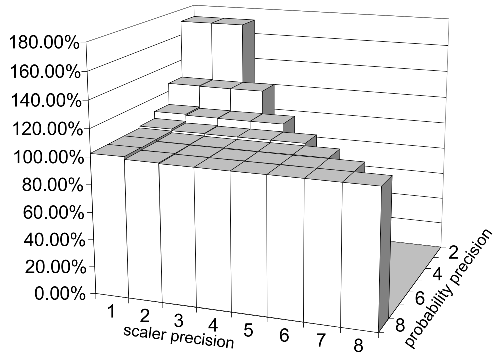

Fig. 2. Effect of precision on compression ratio

the precision of the operands involved, however this reduction results in a penalty in terms of compression ratio.

We have studied the effect of precision reduction on the compression ratio by encoding all data sets available using different operand sizes. The results are presented in Figure 2 and Table 1. The value represents the size of the data compressed with the given precision relative to the ideal compression ratio. The penalty is asymetric with respect to the two operands of the multiplication. A higher precision of the external probability value is more important than the internal scaler. The first column in the table corresponds to using no multiplier at all, a solution that is used in popular image compression schemes.

We have focused our study on low-precision encoding schemes, which allow efficient hardware implementation without a large penalty in terms of compression ratio. A probability representation of 6 bits and scaler of only 3 bits allow achieving a compression ratio of only 0.8% of the theoretical bound.

Arithmetic coding is known to provide optimal compression, subject only to the limitations of the statistical model used to provide the probability of symbol occurrence. When developing the statistical model we keep in mind the hardware requirements of the decompressor.

Table 1. Effect of precision on compression ratio

scaler precision →

	1	2	3	4	5	6	7	8
2	1.710	1.707						
3	1.262	1.256	1.254					
4	1.098	1.087	1.085	1.083				
5	1.045	1.030	1.026	1.025	1.025			
6	1.030	1.013	1.008	1.007	1.006	1.006		
7	1.030	1.010	1.004	1.003	1.002	1.002	1.002	
8	1.031	1.009	1.003	1.001	1.001	1.000	1.000	1.000

probability precision ↓

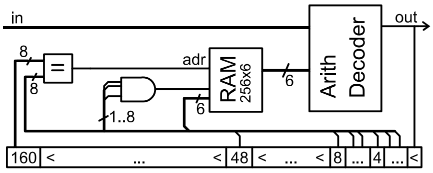

Fig. 3. Decompressor

Our work started by building correlation maps of the bits inside each frame of the bitstream. In order to simplify the structure of the decoder we then limited our search to the correlation of each bit with other bits occupying certain fixed positions relative to itself. Such an approach is advisable as it allows the utilization of a shift register as a history context. By exhaustive search within the length of one frame we determine the subset of bits occupying positions $\{160, 112, 48, 24, 8, 4\}$ in the history, as the most relevant in generating the symbol probability. To further improve the model we set up two special conditions that may affect the probability of the incoming symbol. One tests for runs of consecutive zeros and is implemented as an AND between the last 8 or 16 processed bits. The other determines if the sequence at a displacement of 160 bits matches the current sequence. This model has produced consistent results over all tests and seems to be a characteristic of the FPGA family our tests targeted.

Symbol probabilities are stored in a table which has to be initialized prior to beginning the decompression. The hardware decompressor is presented in Figure 3.

3.2. A Fixed Dictionary Approach

The largest limitation of the decoder in terms of speed is the number of bits it can process at a time. In this respect, compression methods like LZ78 [15] most widely known through its variant LZW, have the distinct advantage of being able to read an entire input word at a time, as encoded words have the same length. However, the same technique has the disadvantage of having to dynamically generate and maintain the contents of the dictionary.

A solution that targets both speed and simplicity would

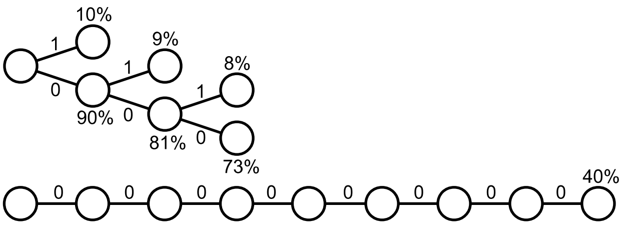

Fig. 4. Symbols for fixed dictionary compression

be to use a statical dictionary that is computed based on the contents of the entire bitstream and is used throughout the entire decompression. Unlike the Huffman dictionary, there is no clear methodology for how such a dictionary can be created in an optimal way (at least not to the knowledge of the authors), but the characteristics of the bitstream make the choice an easy one. In particular, due to the high probability of occurrence of the zero symbol, the coding scheme degenerates into a bit-level RLE with minor modifications.

In order to make sure that there is always a way to encode any input sequence, we build our dictionary in a way similar to the Huffman tree. Starting from the root, we add two branches, one corresponding to the sequence formed of one ”0” bit and one for the sequence consisting of one ”1” bit. After that, as long as there are enough codes to represent more sequences, we expand the most commonly occurring sequence that we already have a representation for, by adding to it two new branches, the same way as the first step. Additional symbols that are not part of the tree can be used as shortcuts. Figure 4 illustrates a possible dictionary composition. The stored sequences are obtained by traversing the tree edges from the root to the leaf nodes, or the extra symbol chains from left to right. The percentages marked on the figure are probabilities of occurrence of the sequences ending on the specific nodes.

Unfortunately, the size of the dictionary grows exponentially with the word size chosen for the encoding, so we only found it feasible to use a word size of 4. Unlike most of the word-based compression methods, but similar to 1-bit LZW, the method does not take advantage of the natural data alignment, which results in a penalty in terms of compression ratio.

4. EXPERIMENTAL RESULTS

The compression algorithms that were evaluated are: the commercial software RAR (used with the highest compression setting), the arithmetic coding and the fixed dictionary method described in Section 3, Huffman coding [16], the dictionary based methods LZSS [17] and LZW [15], the Burrows-Wheeler transform [18], and the combination of Huffman and LZSS. We have used our own implementations for all algorithms except RAR, in order to be able to test various word size as suggested in previous studies and we tested the implementations by correctly decompressing the output.

4.1. The benchmark bitstreams

In our experiments we have assumed large designs in order to ensure a high area utilization of the FPGA. The tests were produced using the default configuration of the hdl synthesizer (Xilinx ISE 6.3i), with no attempt to increase structure regularity by manual optimization. All tests were performed using bitstreams for an xc4vlx25 Virtex 4 FPGA, which have a size of approximately 1MB.

We utilized a benchmark suite composed out of eight designs. Five were real-world tests mostly originating from opencores.org: a general purpose processor (opnrisc), a floating point unit (fpu), a dataflow processor (dflow), an array of AES encoders and decoders (aes), and an array of Ethernet controllers (ether). Of the last two, as many instances were created as would fit on the chosen FPGA chip. Three other tests were automatically generated to use the FPGA structure at the maximum possible extent: a perfectly regular mesh of look-up tables (mesh), a circuit with random connections (randlnk), and a circuit with random connections and forcedly placed to uniformly cover the surface (randfull). For the mesh circuit perfect regularity was ensured by connecting each LUT to four of its neighbors, having in all cases the same relative placement. At the edges of the mesh, wrap-around occurred. In spite of this regularity we found that synthesis tools have a randomizing effect in both placing and routing, which resulted in little similarity to be exploited by the dictionary based compression algorithms.

We have only considered the useful bitstream data for compression, empty frames were discarded as they would have generated unrealistically good reports. Error detection codes were also discarded, since it would make more sense to send them uncompressed.

4.2. Compression Efficiency

Here, we present the compression efficiency of the evaluated algorithms using the favorable word lengths. The names present in the table are as follows: “rar” is of course the RAR commercial software, “arith” is the arithmetic coder using a simple statistical model that only takes into account the ratio of 0 and 1 bits, “apc6x3” is the arithmetic coder using the statistical model presented in Section 3, “apc5x2r” is an arithmetic coder with reduced precision and a simplified statistical model still based on the one described in Section 3, “huf4” and “huf8” are the Huffman encodings with word sizes of 4 and 8 respectively, “lzw4” is the LZW algorithm with a word-size of 4, “bwt8” is the Burrows-Wheeler transform followed as suggested by the authors by move-to-front

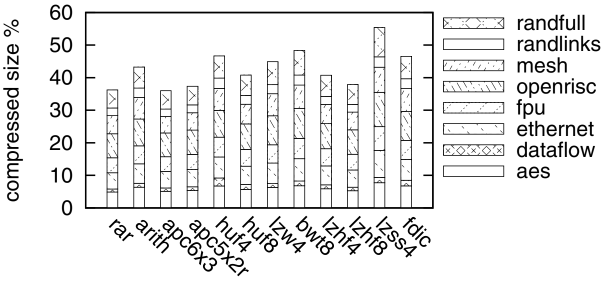

Fig. 5. Compression efficiency

Table 2. Compression efficiency

	aes	dflow	ether	fpu	opnrisc	mesh	randlnk	randfull
rar	33.93%	11.28%	34.90%	32.24%	51.00%	42.23%	33.28%	39.13%
arith	44.25%	14.87%	41.32%	39.18%	56.67%	50.29%	41.35%	45.60%
apc6x3	35.72%	12.07%	35.56%	31.58%	50.48%	38.50%	32.44%	39.68%
apc5x2r	37.16%	14.27%	36.53%	32.81%	51.71%	40.40%	34.45%	40.35%
huf4	46.89%	29.87%	44.64%	43.10%	56.62%	51.47%	45.13%	48.04%
huf8	39.47%	19.09%	38.45%	36.40%	54.46%	45.51%	38.97%	43.95%
lzw4	44.10%	13.89%	43.66%	39.87%	61.62%	51.38%	39.58%	49.73%
bwt8	47.70%	17.13%	47.60%	44.01%	64.00%	54.07%	43.73%	53.27%
lzhf4	41.13%	14.50%	40.21%	37.35%	53.36%	44.59%	35.02%	45.89%
lzhf8	37.04%	12.76%	36.43%	34.17%	52.09%	41.26%	33.92%	43.01%
lzss4	54.05%	19.85%	57.53%	51.25%	72.90%	58.37%	45.92%	63.40%
fdic	47.12%	20.91%	44.27%	41.37%	61.36%	53.61%	43.01%	48.39%

and Huffman, "lzhf4" and "lzhf8" represent the LZSS compression method followed by Huffman coding, "lzss4" is the plain LZSS method and finally "fdic" is the fixed dictionary approach.

All methods were tested using the 8 benchmarks previously mentioned. The results, expressed as a ratio between the compressed size and the initial size (excluding zero frames and error recovery codes), are presented in Table 2 and plotted on the graph in Figure 5. The leading algorithm in terms of compression ratio is the arithmetic coding used in conjunction with our statistical model.

4.3. Word length

Most of the compression algorithms are sensitive with respect to the size of the encoded words. Previous studies [9, 8] have shown word sizes of 6 and 9 to be more effective for compressing the bitstreams of Virtex (1) FPGAs. This was based on knowledge of the internal organization of the bitstream and was verified experimentally. As the internal structure of the Virtex-4 family bitstreams is not disclosed by the manufacturer, we have performed our tests for all word sizes within the feasible range. The results of the experiment are presented in Figure 6. Local minima can be observed for word sizes of 4 and 8 bits.

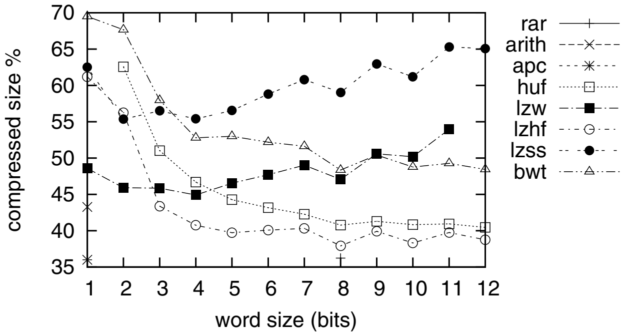

Fig. 6. Effect of word size on compression ratio

Table 3. Performance vs. Cost

	ratio	compr. penalty	area (slices)	clock (MHz)	troughput (Mbps)
rawsize	100.00%	177.82%	0		
arith	43.27%	20.20%	89	334	334
apc6x3	36.00%	0.00%	153	198	198
apc5x2r	37.36%	3.79%	105	351	351
huf4	46.68%	29.67%	14	424	908
huf8	40.77%	13.27%	12+1bram	306	751
lzhf4	40.76%	13.24%	83	228	479
lzhf8	37.91%	5.31%	117+3bram	227	584
lzss4	61.04%	69.58%	60	277	926
fdic4	46.54%	29.30%	115	395	3397

4.4. Hardware implementations

For the compression algorithms showing good compression ratios we provide hardware implementations. Table 3 includes these results along with the respective compression ratios. We implemented arithmetic modules with and without prediction, Huffman decoders, a LZSS decoder, a fixed dictionary decoder, and a combined LZSS and Huffman decoder. The rawsize entry represents the bitstream size without compression.

The decoders were implemented in Verilog and synthesized the Xilinx ISE 9.2i suite using settings for speed optimization. The decoders were pipelined in order to achieve better performance. The presented values for area and frequency are post place and route, as reported under "Best Acheivable Case". The speed grade selected was 11, with default values for voltage and temperature. Verification was performed in simulation post-synthesis.

For the LZHF8 decoder a large, 4KB dictionary was used as the module already had large memory requirements because of from the Huffman decoder, while for LZHF4 and LZSS a smaller dictionary of 32 words was used. RAR was not included in the table as it is not suitable for a hardware implementation. The compression penalty is expressed in terms of a percentage of the size increase relative to the file produced by the most efficient algorithm.

The highest advantage in terms of speed, 266% above the next competitor, belongs to the fixed dictionary approach. The drawback is a relatively low compression ratio. At the other end, the arithmetic coder achieves the highest compression ratio at a cost in area and more importantly speed (a loss of 78.6% in speed). Two of the decoders, those using large Huffman tables, require the usage of Block RAMs in addition to FPGA slices.

The frequencies obtained for the decoders range from 198Mhz to 424Mhz, however, the hardware implementations of different algorithms are able to produce a different number of output bits per clock. Consequently, the speed of the decoders was expressed in effective output rate rather than clock frequency.

Buffering circuitry and serializers/deserializers were not included with the exception of the fixed dictionary decompressor which had special requirements because of its high throughput. Decoders that have either a steady input or a steady output rate, i.e., all decoders except LZHF, have an advantage by requiring buffering at one end only. In addition, Arithmetic and Huffman coding have the advantage of a more steady compression ratio, while LZSS is at the opposite end, producing bursts when a sequence already in the dictionary is found.

5. CONCLUSIONS

In this paper we have studied the opportunity of using compression for accelerating configuration and reconfiguration of FPGAs. The contributions of this paper can be summarized as follows: we implemented a wide range of compression algorithms, for variable word widths; we implemented highly optimized hardware decompressors for those algorithms showing promising results; we have tested techniques previously shown to provide an advantage in compressing FPGA bitstreams; we have proposed two new compression methods, specifically tailored to the purpose of FPGA bitstream compression.

Our study suggests that the internal organization of bitstreams is likely to change from one family of FPGA devices to another. This was found true when comparing the Virtex 4 with the devices targetted by previous studies. At least one conclusion though seems likely to hold in the future: synthesis tools produce a randomization of the input bitstream, leaving the ratio of "0" and "1" symbols as a main source of redundancy and turning the focus toward simple compression methods.

6. REFERENCES

[1] S. Hauck, Z. Li, and E. Schwabe, "Configuration compression for the Xilinx XC6200 FPGA," in *IEEE Symposium on FPGAs for Custom Computing Machines*, 1998, pp. 138–146.

[2] Z. Li and S. Hauck, "Don't care discovery for FPGA configuration compression," in *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, 1999, pp. 91–98.

[3] J. H. Pan, T. Mitra, and W.-F. Wong, "Configuration bitstream compression for dynamically reconfigurable FPGAs," in *International Conference on Computer Aided Design (ICCAD)*, November 2004, pp. 766–773.

[4] ALTERA Corporation, *Stratix II Device Handbook, Volume 2*, 2005.

[5] Altera Corporation, *Enhanced Configuration Devices (EPC4, EPC8 & EPC16) Data Sheet*, October 2005.

[6] Xilinx Inc., *System ACE MPM Solution*, June 2003.

[7] ——, *Virtex-4 User Guide*, February 2005.

[8] M. Huebner, M. Ullmann, F. Weissel, and J. Becker, "Real-time configuration code decompression for dynamic FPGA self-reconfiguration," *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS04)*, pp. 138–143, 2004.

[9] Z. Li and S. Hauck, "Configuration compression for Virtex FPGAs," *Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 147–159, 2001.

[10] S. Hauck and W. D. Wilson, "Runlength compression techniques for FPGA configurations," in *IEEE Symposium on FPGAs for Custom Computing Machines*, 1999, pp. 286–287.

[11] F. Farshadjam, M. Fathy, and M. Dehghan, "A new approach for configuration compression in Virtex based RTR systems," in *Canadian Conference on Electrical and Computer Engineering, 2004*, vol. 04, 2004, pp. 1093– 1096.

[12] A. Dandalis and V. K. Prasanna, "Configuration compression for FPGA-based embedded systems," in *FPGA '01: Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, 2001, pp. 173–182.

[13] M. Martina, G. Masera, A. Molino, F. Vacca, L. Sterpone, and M. Violante, "A new approach to compress the configuration information of programmable devices," in *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, May 2006, pp. 48–51.

[14] J. Rissanen and G. G. Langdon, Jr, "Arithmetic coding," vol. 23, no. 2, pp. 149–162, Mar. 1979.

[15] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, 1978.

[16] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, September 1952.

[17] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[18] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm., Tech. Rep. 124, 1994.