ZipStream: improving dependability in Dynamic Partial Reconguration

ZipStream: improving dependability in Dynamic Partial Reconfiguration

Stefano Di Carlo, Giulio Gambardella, Trong Huynh Bao*, Paolo Prinetto, Daniele Rolfo, Pascal Trotta
Politecnico di Torino
Dipartimento di Automatica e Informatica
Corso Duca degli Abruzzi 24, I-10129, Torino, Italy
Email: {name.familyname}@polito.it
*Email: trong.huynhbao@studenti.polito.it

***Abstract*—Dynamic Partial Reconfiguration allows to dynamically change the behaviour of a portion of the FPGAs by downloading new information in the configuration memory of the device. Since modern Systems-on-Programmable-Chips (SoPCs) make extensive use of this feature, many reconfigurable area are placed in the device, with several configurations for each area. This comes at a cost in terms of dependability of the system and of memory occupation. The proposed methodology focuses on increasing the dependability of partially reconfigurable systems by safely storing compressed configuration data inside the FPGA.**

I. Introduction

Field Programmable Gate Arrays (FPGAs) are devices composed of interconnected functional blocks that can be programmed to accomplish a desired logic function.

Because of their programmable nature, they have often been used for prototyping and debug [1]. With technology shrinking and technical improvements, the performance gap between FPGAs and Application Specific Integrated Circuits (ASICs) has progressively decreased [2]. As a result, FPGAs are now commonly used also in final release implementations, primarily in small quantities and ad-hoc custom products, like space applications [3] [4].

The use of FPGAs in safety critical systems introduces new challenges in the design phase. Nevertheless, their flexibility allows the designer to devise innovative methodologies to increase the system dependability. Dynamic Partial Reconfiguration (DPR) is a widely used feature, offered by Xilinx FPGAs. DPR gives the designer the capability to reconfigure a certain portion of the device at run-time, without influencing the other portions of the design.

The partial reconfiguration process consists of downloading the information listed in the *partial bitstream* in the configuration memory, through the *Internal Configuration Access Port* (ICAP) [5]. At the end of the reconfiguration, the reconfigured partition is able to accomplish the desired function. In a System-On-Programmable-Chip (SoPC) requiring a high level of flexibility, many reconfigurable partitions can be inserted. Each partition can be exploited to time-multiplex the same hardware resources for different functional modules, by loading configurations at different times. This significantly increases the number of partial bitstreams to be stored.

Partial bitstreams may be stored either inside the FPGA or in an external memory.

Storing the partial bitstream inside the FPGA provides a high level of dependability, since the Block RAMs (BRAMs) (i.e., internal memory in Xilinx' FPGA) in the device are, in general, protected by correction codes [6], and the connection to the ICAP is safer then the one to external devices. However, this high dependability comes at the cost of an increased internal memory occupation, that, in many designs, proved to be a very critical issue.

Storing configuration data in an external memory enables to store many partial bitstreams. However, the external connections increase the probability of reading faulty data.

To solve the resources utilization problem, many approaches have been proposed to compress the partial bitstream data in order to decrease the memory occupation in the FPGA [7] [8] [9] [10] [11]. Most of them use lossless data compression schemes, like Lempel-Ziv based [12] [13]. Since they require very smart and big hardware decompressor, these approaches clearly led to a decrease in terms of available logic resources.

In the proposed paper we describe ZipStream, an innovative methodology to improve the dependability in dynamically reconfigurable systems by backuping compressed partial bitstreams.

For each reconfigurable partition, a compressed partial bitstream is stored internally. To achieve a very high compression ratio and a very low usage of resources for the decompressor, a special partial bitstream, called *black-box*, is exploited. If a reconfiguration process fails, the non-reconfigurable portion of the system may be damaged, too. The reconfigurable partition is then configured with the associated *black-box* partial bitstream. This reconfiguration restores the non-reconfigurable system functionalities and results in a system graceful degradation, in which the non-reconfigurable portion of the SoPC is still working, even if the reconfigured partition will not be able to provide the original desired function.

The paper is organized as follows: Section II introduces the DPR and the related dependability issues. Section III briefly introduces compression algorithms. Section IV describes the ZipStream methodology, while experimental results, gathered targeting the Xilinx FPGA architecture, are shown in Section V. Section VI concludes the paper.

II. Dynamic Partial Reconfiguration

Dynamic Partial Reconfiguration (DPR) extends the native FPGA flexibility. It enables to dynamically change functionalities of a section of a circuit (Partition), while the rest of the design is left unchanged and fully functional. Therefore, the ability to time-multiplex hardware modules at run-time enables to design complex systems reducing cost, board space, and power consumption.

The partial reconfiguration process can be activated at run-time by downloading a partial bitstream inside the FPGA through a configuration port (e.g., ICAP in Fig. 1). These information are written inside the configuration memory of the SRAM-based FPGA.

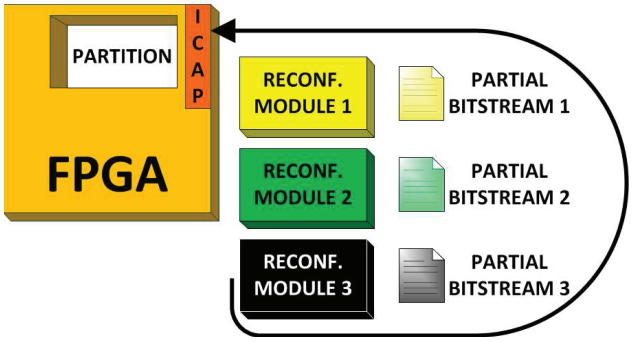

Figure 1: Dynamic Partial Reconfiguration scheme

The ICAP provides the highest bandwidth (i.e., 3.2 Gbps [14]), speeding up the reconfiguration process.

After the whole partial bitstream has been downloaded in the ICAP, the addressed section of the configuration memory is modified. These modifications change the behavior of the reconfigurable partition. Unfortunately, this alteration of the configuration memory can introduce some dependability issues. In fact, even if the changes address just the reconfigured partition, an error occurring during the configuration memory writing may affect the proper functioning of the static part of the design, as well. This because when a partial reconfiguration design is placed and routed, static routes (i.e., routes associated to non-reconfigurable portion of the design) can route through reconfigurable partitions [5].

When the partial bitstream is corrupted, some connections can be broken, entailing critical system failures. In such a case a complete reconfiguration of the FPGA is needed to restore the system.

The ICAP provides a built-in CRC check at the end of the reconfiguration process, in order to check if the downloaded bitstream is corrupted. This "late" check doesn't provide any protection to the system, since the corrupted reconfigurable partition is already active, potentially damaging the device.

A different approach must therefore be added at design time to prevent faulty reconfigurations. Xilinx, in the Partial Reconfiguration User Guide [5], takes care of the partial bit file integrity. In particular, the proposed solution requires both a software and a hardware layer [5].

This solution guarantees, also for faulty reconfigurations, that the rest of the system (i.e., static portion) continues to function. Nevertheless, it introduces an amount of latency due to the adopted buffering mechanisms that, in some cases, is not acceptable.

In addition, the recovery mechanism is not described by Xilinx, and the choice is left to the user. A widely used approach consists in reading again the bitstream and trying to reconfigure again with the same bit file. This process must be handled by a reconfiguration manager and leads to a very long reconfiguration time. Moreover, it does not assure that the reconfiguration process will finally ends properly.

In fact, if the bitstream read from the external memory is definitely corrupted, the reconfiguration process will always be stopped.

A widely used more dependable solution requires to store the bitstream inside the FPGA. Since the BRAMs of the device are protected by Error Correction Codes (ECCs), the bitstream will be safely stored. Nevertheless, this will come at a cost in terms of memory occupation, that in modern SoPCs is a really critical resource. An effective solution is to compress the data to be stored, in order to reduce the memory occupation.

The next section presents an overview of compression algorithms, taking into account the complexity required for the hardware implementation.

III. Compression Algorithms overview

A compression algorithm aims at decreasing the size of an original array of data, involving encoding information. The achieved compression can be either lossy or lossless.

Lossy algorithms reduce the dimension of the array by eliminating marginal information. Usually, a lossy compression achieves a high compression rate, but the original data can not be reconstructed. Examples of lossy compressions are MPEG2, JPEG and MP3 [15].

Lossless compression algorithms allow the original array to be reconstructed from the compressed data. Lossless data compression is usually based on the statistical model of the input data, exploiting redundancy to compress the original data. Therefore, the achieved compression rate is closely related to the statistics of the data to be compressed.

Two main lossless encoding algorithms are used: *Huffman Coding* and *Arithmetic Coding*. In both cases, the most frequently used characters are coded by fewer bits, while less frequently occurring characters are coded by more bits. This led to fewer bits used in total.

Huffman coding [16] is the most well-known and widely used variable-length code. It separates the input data into symbols (choosing their lengths) and replaces each symbol by a code. Arithmetic coding, instead, encodes the entire message into a single number. Arithmetic coding [17] achieves high compression rates for particular statistical models, whereas it implies higher computational complexity. Huffman compression is simpler and faster, but produces poor results for models that deal with uniform symbol probabilities. It provides very high

compression ratios when input data are very redundant.
Another kind of compression algorithm, based on the characteristics of the data to be compressed, is the *Run-length encoding* (RLE) [16]. In RLE, sequences of the same value are stored as a single data and the number of occurrences in the sequence.

IV. ZipStream

ZipStream is a novel methodology to increase the level of dependability in reconfigurable FPGA systems by internally backuping special compressed bitstreams. The solution is based on storing in the FPGA a compressed partial bitstream for each reconfigurable partition. When a corrupted bitstream (i.e., checked by the ICAP built-in CRC) is read from the external memory, at the end of the reconfiguration the static part of the system may be damaged (see Section II). In order to restore in a very short time the static system functionalities, an internally stored *black-box* [18] bitstream configures the corrupted reconfigurable partition.
Since the main goal is to assure the proper functioning of the system, the compressed bitstream will reconfigure the target partition with a *black-box* module, able to ensure the static connections passing through the reconfigurable partition to be correctly restored.
The *black-box* partial bitstreams, generated by the Xilinx PlanAhead tool, encompass just the information of the static part of the reconfigurable partition, since there is neither logic nor nets associated with this partition. The main peculiarity of this special bitstream is that it embedds a lot of data redundancy exploitable for an efficient compression.
In general, a partial bitstream contains information items related to the logic and routing resources present in the associated reconfigurable partition. A *black-box* partial bitstream carries just the information about the static connections of the design which routes through the targeted reconfigurable partition. Since it does not contain any information associated with the logic resources, it includes long sequences of zeros. Since the ICAP CRC check does not occur until the end of a reconfiguration process, a faulty bitstream loaded from the external memory may damage the static portion of the design, including connections associated to the reconfiguration controller. In this case, the system will not be able to restore the faulty partition by a *black-box* module. Thus, to make the ZipStream methodology effective, a *Design-for-Dependability* (DfD) rule must be adopted at design time.
In addition, the features of the *black-box* partial bitstreams enable the use of an ad-hoc designed compression algorithm based on the Huffman encoding (see Section IV-A), whose decompressor requires very low hardware resources.
The ZipStream approach consists of three different steps:

- apply the optimal *Compression algorithm* to the partial bitstreams in software;
- design the *Hardware Decompressor* to be implemented in hardware;
- apply the *DfD* design rule to the system design.

A. Compression Algorithm

The partial bitstream file is a stream of data to be downloaded in the ICAP, composed of hundreds of 32-bit words. To compress these data, a combination of two different algorithms has been used to achieve a high compression rate.
Since the bitstreams to be compressed are characterized by many sequences of zeros, the first applied algorithm was the *Run-Length Encoding* (RLE).
The RLE encodes a sequence of consecutive zeros by a binary number. Obviously, the insertion of 1-bit flag to distinguish normal symbol or encoded symbol is required. This encoding is based on splitting the stream of bits in symbols. All symbols have the same length in terms of number of bits. The length clearly has a great impact on the compression ratio, since the symbol probability changes with the symbols' length. In order to enhance the flexibility of the methodology, the software compressor is able to automatically calculate the symbol length to achieve the best compression rate. To do this, the RLE algorithm is applied with three different symbol lengths (e.g., 8, 16 and 32 bits). The set of lengths has been chosen to better fit the data in the hardware, since the decompressor should save these data in a BRAM, 8-bit words addressable.
After applying the RLE algorithm, the software generates the compressed bitstreams, using the best symbol length configuration.
Afterwards, the Huffman encoding is applied on the RLE encoded bitstreams. Huffman coding has been chosen by evaluating the hardware resources needed in the decoder for both Huffman and Arithmetic coding. In fact, the complexity of the Huffman decoder is definitely lower than the Arithmetic one.
The Huffman encoding process consists of two steps: *Huffman tree construction* and *Single side Growing Huffman encoding*. The *Huffman tree construction* implements the following algorithm: occurrences of all words are evaluated, then the array of words is sorted in descending order of frequency, in order to correctly generate the Huffman tree. This frequency-sorted binary tree is then used to assign the symbols to the words in the standard Huffman encoding. The more frequent is the word, the shorter will be the assigned code.
Nonetheless, since these data must be stored inside the FPGA BRAMs, the trade-off between performances and storage occupation need to be considered carefully. Considering the hardware implementation, the data structures should not be too complex to reduce the logic area occupation.
In this context, the Single-side Growing Huffman (SGH) tree [19] proved to be the most efficient in terms of memory occupation [20]. The reader may refer to [20] for more information.
The previously constructed Huffman tree is translated into a SGH tree.
In the proposed methodology the SGH tree coding process is fully computed by software, providing as output the compressed bitstream. Table I is an example of Single-side Growing Huffman table, in which the symbols are listed in

order of code length. The translation table must be stored

Table I: Single-side Growing Huffman table

Symbols	Code
s1	0
s2	1
s3	1000
s4	1001
s5	1010
s6	1011
s7	1100
s8	1101
s9	11100
s10	111010
s11	111011
s12	111100
s13	111101
s14	111110
s15	111111

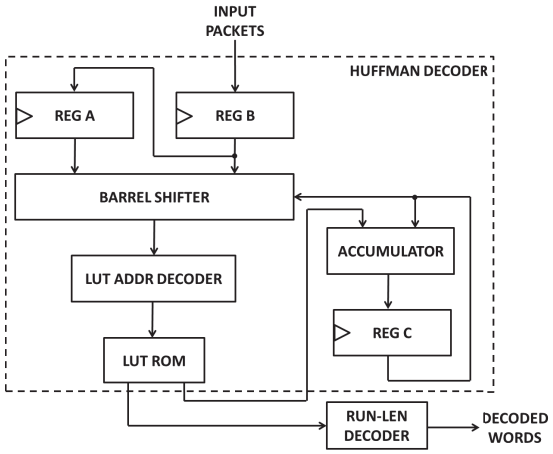

Figure 2: Decompressor architecture

inside the FPGA internal memory. The memory occupation has been reduced thanks to a smart approach based on the special features of the SGH table. In fact, it is easy to group the code words depending on the number of the prefix ones. This introduces the possibility to decode the code words in a hierarchical way (see Sec. IV-B and Sec. V).
The hierarchical decoding is done in two different steps. Code word groups can be simply identified by using hardwired logic, while symbols decoding is done exploiting a *Look-Up-Table* approach. As example, when symbols length is 8 bits, and the code length is 12 bits, the original memory size needed to store the LUT is 4 KB. In our solution, the LUT size is only 128 B.
After the coding algorithm outputs the code word for each symbol, the *Look-Up-Table* (LUT) ROM is generated to store the data in the FPGA.

B. Hardware Decompressor

The decompressor (or decoder) is hardware implemented in the device logic. The decompressor performs the Huffman decoding on the input code words, and then the RLE, if necessary, providing in output the original bitstream data. Fig. 2 shows the architecture of the hardware decoder.
Initially, the 16-bit value of the *REG C* register, that represents the number of already decoded bits of the input packet, is 0. The 16-bit packet received in input is stored in *REG B* register. The packet is passed directly to the *LUT ADDR DECODER*, without performing the shift operation. The *LUT ADDR DECODER* decodes the group which the code word belongs to, by counting the number of leading 1 in the code word. After identifying the group, the *LUT ADDR DECODER* generates the address for the *LUT ROM*, which contains the SGH table.
The *LUT ROM* outputs the symbol associated to the input code word and the relative code length. The symbol is then passed to the *RUN-LEN DECODER*, while the code length is accumulated by the 16-bit *ACCUMULATOR* and the result is stored in the *REG C* register. The value of this register controls the *BARREL SHIFTER* shift operations.
The *BARREL SHIFTER* module shifts the input packet of the number of bits indicated by *REG C*, flushing the already decoded bits. In this way, only the undecoded bits appear at the input of the *LUT ADDR DECODER*.
In case there is an overflow in the *ACCUMULATOR*, it means that a codeword is split between two consecutive packets, so the next packet is received in *REG B* register, while the previous packet is moved to the *REG A* register.
REG A and *REG B* values are concatenated and input to the *BARREL SHIFTER*. The described operations are repeated recursively until no more input packets are received.
The output symbol from the *LUT ROM* is finally decoded by the *RUN-LEN DECODER*, if necessary. This decoder checks the flag bit in symbols. Whenever the flag is asserted, it implies that this symbol has to be decoded by the *RUN-LEN DECODER*. The *RUN-LEN DECODER* read the binary value of the symbol and output a sequence of consecutive zeros with a length equal to that value. The output of the *RUN-LEN DECODER* represents the decoded words of the bitstream, that can be passed to the ICAP interface exploiting the reconfiguration controller.
The operations of the hardware decompressor are managed by the reconfiguration controller. The controller is directly connected to the ICAP interface to download the bitstream with correct timing.
At run time, when a reconfiguration request is received, the controller read an external stored bitstream in order to reconfigure a partition. If an error at the end of the reconfiguration process is detected by the ICAP, the controller starts reading the compressed *black-box* bitstream from the internal BRAM of the FPGA. Then, it manages the decoding process and it provides the decoded words to the ICAP interface.

C. Design-for-Dependability rule

As mentioned above, a corrupted partial bitstream, read from the external memory, can led to a system damage. To avoid the corruption of the connections of the modules used to restore the correct behaviour of the static part of the system, a simple *DfD* rule must be adopted.
The rule aims at enclosing all the aforementioned connections in such a way that they cannot be routed trough a reconfigurable partition. This can be achieved by including in a non-reconfigurable partition all the critical blocks:

- Reconfiguration Controller (RC)
- ICAP
- BRAMs, that store the *black-box* bitstreams
- Decompressor

This rule ensures that all connections between these blocks, that are necessary after a faulty reconfiguration, do not pass through the faulty reconfigured partition. Thus, the correct functioning of these modules is guaranteed also after a faulty reconfiguration.
Fig. 3 shows an example of how these blocks can be protected. The highlighted block on the top of the device represents

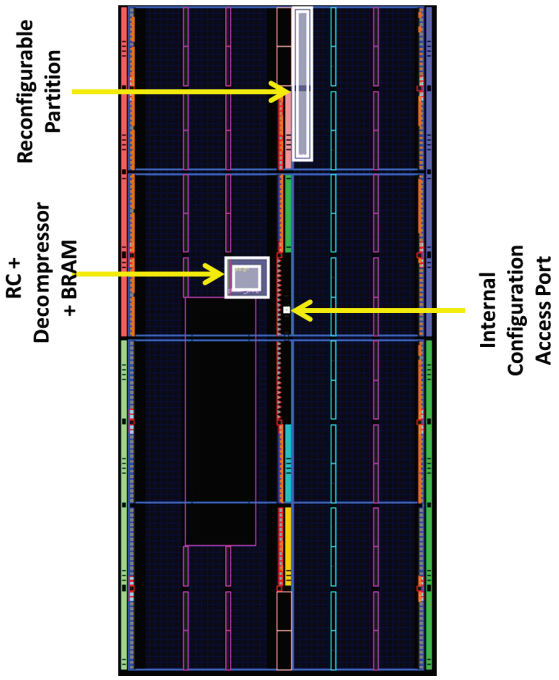

Figure 3: Example of protecting the critical blocks

the reconfigurable partition. The non-reconfigurable partition, in the middle of the device, includes the Reconfiguration Controller, the *Hardware Decompressor*, and the BRAM that stores the *black-box* bitstream associated to the reconfigurable partition. The ICAP cannot be included in a partition, thus it is advisable to verify that the few connections between this interface and the RC do not pass through the reconfigurable partition.

V. Experimental results

The ZipStream methodology is composed by the Huffman software encoder and the hardware decoder. First of all, many partial bitstreams were generated using Xilinx PlanAhead Tool v14.2, targeting Virtex 4 XC4VFX12-FF668-10C FPGA. The design includes the Leon3 soft-core [21] and many different reconfigurable partitions.
Five different reconfigurable partition sizes (i.e., 1,2,3,4 and 5 frames) have been tested. The black-box module has been placed in 10 different positions in the device, in order to have 10 different partial bitstreams for each partition.
To correctly encode the bitstream file and to find the solution that guarantees the best compression ratio, several compression algorithms have been tried. First, *black-box* partial bitstreams are encoded with the standard Huffman algorithm (Huffman 32, 16 and 8 in Fig. 4 and Fig. 5), splitting in different word length configurations: 32 bits, 16 bits, 8 bits. Then, a combination of Huffman and Run-length encoding is used to compress the bitstreams (Huffman+RLE 32, 16 and 8 in Fig. 4 and Fig. 5).
Fig. 4 and Fig. 5 shows the compression rate in different scenarios. The graph plots average compression ratio, between the 10 partial bitstreams considering 5 different partition dimensions. The *combination* column, in Fig. 5, shows the average compression rate in the 50 different bitstreams.
The compression rate is computed as the ratio of compressed data plus the LUT (i.e., SGH tree) size over the original data.

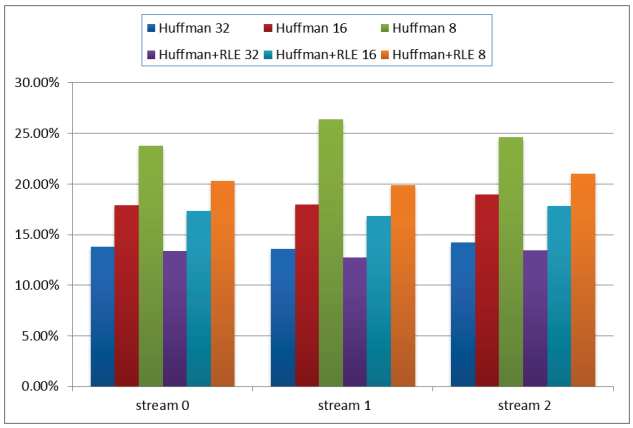

Figure 4: Compression rates for streams 0, 1 and 2

Easily to conclude from Fig. 4 and Fig. 5 that, when word length configuration for Huffman encoding is larger than 8 bits, the overhead of Run-length encoding contributes a portion in the encoded bits and does not bring any benefit. However, the combination of Huffman encoding and Run-length encoding has a good impact when bit length is 8 bits. On overall, the compression rate is around 22% in all cases, it means that we could save 78% of storage size. The best compression rate that can achieved by the proposed solution is 19%, and the best choice is the combination of Huffman

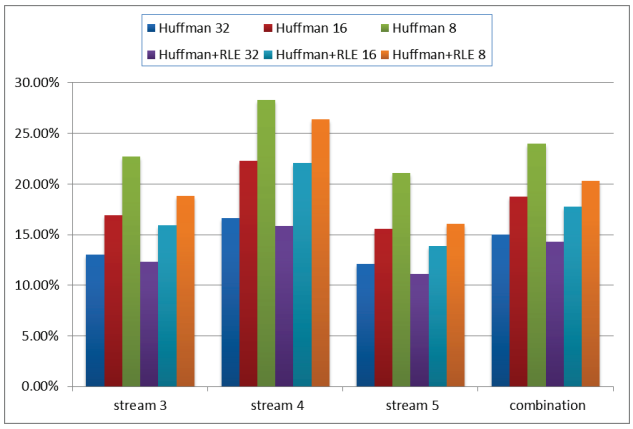

Figure 5: Compression rates for streams 3, 4, 5 and average compression rates

code and Run-length encoding with a word length of 8 bits. The maximum working frequency of the decompressor is 163.55 MHz, with an area occupation of 189 slices (i.e., 3% of the device), including the reconfiguration controller.
Thanks to this implementation, since the ICAP works at 100 MHz, the decoder is able to provide a 32-bit word each clock cycle, i.e., with the same throughput required by the configuration port, reaching the highest reconfiguration data throughput.
Finally, to prove the effectiveness of the ZipStream methodology different faulty bitstreams have been stored in an external memory and loaded to the ICAP. The internal 32 bit CRC module, which polynomial is *0x8F6E37A0*, is able to detect 3 faults in each bitstream [22]. In all cases the reconfiguration controller was able to restore the proper function of the static portion of the system, after the error notification asserted by the ICAP built-in CRC.

VI. Conclusion

This paper presented ZipStream, an innovative methodology for increasing dependability in partially reconfigurable systems. The proposed methodology enhances the dependability of the system by storing a compressed bitstream inside the device as backup. This partial bitstream guarantees the proper behaviour of the whole system by reconfiguring the partition with a blank module, that perhaps correctly routes the static connections through the reconfigurable partition. The optimal compression algorithm has been studied by evaluating both compression ratio and area occupation. The experimental results clearly show that this methodology has no impact on performances and allows a fast reconfiguration process.

Acknowledgment

The authors would like to express their sincere thanks to the whole design team of Ansaldo STS for their helpful hints and guidelines.

References

[1] O. Melnikova, I. Hahanova, and K. Mostovaya, "Using multi-FPGA systems for ASIC prototyping," in *CAD Systems in Microelectronics, 2009. CADSM 2009. 10th International Conference - The Experience of Designing and Application of*, pp. 237–239, feb. 2009.
[2] C. Valderrama, L. Jojczyk, P. Possa, and J. Gazzano, "FPGA and ASIC convergence," in *Programmable Logic (SPL), 2011 VII Southern Conference on*, pp. 269–274, April 2011.
[3] S. Di Carlo, A. Miele, P. Prinetto, and A. Trapanese, "Microprocessor fault-tolerance via on-the-fly partial reconfiguration," in *Test Symposium (ETS), 2010 15th IEEE European*, pp. 201–206, May 2010.
[4] B. Osterloh, H. Michalik, S. Habinc, and B. Fiethe, "Dynamic partial reconfiguration in space applications," in *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on*, pp. 336–343, August 2009.
[5] Xilinx Corporation, *Partial Reconfiguration User Guide*, October 2010.
[6] Xilinx Corporation, *Virtex-6 FPGA Memory Resources*, April 2011.
[7] R. Stefan and S. Cotofana, "Bitstream compression techniques for Virtex 4 FPGAs," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pp. 323–328, September 2008.
[8] J. H. Pan, T. Mitra, and W.-F. Wong, "Configuration bitstream compression for dynamically reconfigurable FPGAs," in *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*, pp. 766–773, November 2004.
[9] H. Gu and S. Chen, "Partial reconfiguration bitstream compression for Virtex FPGAs," in *Image and Signal Processing, 2008. CISP '08. Congress on*, vol. 5, pp. 183–185, May 2008.
[10] M. Huebner, M. Ullmann, F. Weissel, and J. Becker, "Real-time configuration code decompression for dynamic FPGA self-reconfiguration," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, p. 138, April 2004.
[11] 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 *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, vol. 2, p. 4 pp., March 2006.
[12] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *Information Theory, IEEE Transactions on*, vol. 23, pp. 337–343, May 1977.
[13] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *Information Theory, IEEE Transactions on*, vol. 24, pp. 530–536, September 1978.
[14] Xilinx Corporation, *Partial Reconfiguration of Xilinx FPGAs Using ISE Design Suite*, July 2011.
[15] J. L. Mitchell, W. B. Pennebaker, C. E. Fogg, and D. J. Legall, eds., *MPEG Video Compression Standard*. London, UK, UK: Chapman & Hall, Ltd., 1996.
[16] H. Books, *Articles on Coding Theory, Including: Huffman Coding, Run-Length Encoding, Bch Code, Hamming Code, Hamming Distance, Reed "Solomon Error Correction, Prefix Code, Binary Symmetric Channel, Unary Coding, Low-Density Parity-Check Code*. Hephaestus Books, 2011.
[17] G. G. Langdon, "An introduction to arithmetic coding," *IBM J. Res. Dev.*, vol. 28, pp. 135–149, Mar. 1984.
[18] Xilinx Corporation, *PlanAhead User Guide*, ug632 (v11.4) ed., December 2009.
[19] R. Hashemian, "Memory efficient and high-speed search huffman coding," *Communications, IEEE Transactions on*, vol. 43, pp. 2576–2581, October 1995.
[20] Y.-J. Chuang and J.-L. Wu, "An SGH-tree based efficient huffman decoding," in *Information, Communications and Signal Processing, 2003 and Fourth Pacific Rim Conference on Multimedia. Proceedings of the 2003 Joint Conference of the Fourth International Conference on*, vol. 3, pp. 1483–1487 vol.3, dec. 2003.
[21] J. Gaisler, "A portable and fault-tolerant microprocessor based on the SPARC v8 architecture," in *Dependable Systems and Networks (DSN), 2002. International Conference on*, pp. 409–415, June 23–26, 2002.
[22] P. Koopman, "32-bit cyclic redundancy codes for internet applications," in *Dependable Systems and Networks (DSN), 2002. International Conference on*, pp. 459–468, December 2002.