Generic Scrubbing-based Architecture for Custom Error Correction Algorithms

Rui Santos, Shyamsundar Venkataraman
Department of Electrical & Computer Engineering
National University of Singapore
Email: {elergvds, shyam}@nus.edu.sg

Akash Kumar
Center for Advancing Electronics Dresden
Technische Universität Dresden
Email: akash.kumar@tu-dresden.de

***Abstract*—Scrubbing has been considered as an efficient mechanism to repair faults in the FPGA's configuration memory, when they are placed in harsh environments. By using this elementary mechanism, several academic solutions/algorithms based on error correction codes (ECCs) have been proposed. However, most of these proposed solutions are only theoretical and do not properly deal with the implementation concerns. With this paper we propose a generic scrubbing-based hardware architecture and design flow for implementing custom error correction algorithms based on ECCs. A conducted case study implementing and evaluating three different algorithms shows the feasibility and the efficiency of the proposed architecture and design flow.**

I. Introduction

Nowadays, embedded systems play a central role in advanced and critical technological areas such as transport (avionics, aerospace, automotive, trains), telecommunication (satellites, mobile phones, network components), process control (industrial automation, energy production and distribution) and health (health monitoring, surgery assistance) sectors. This diversity in applications and requirements has been leading to an increase in the complexity of these embedded systems and consequently to a growing demand for increased performance, parallelization and signal processing.

As a response to these concerns, commercially-off-the-shelf (COTS) Static-RAM (SRAM)-based Field Programmable Gate Arrays (FPGAs) have been gaining a greater role in the embedded systems development. FPGAs offer a high operational capacity and performance combined with reconfigurable properties. They also offer a flexible interface, fast I/O response, and an easy way to implement specific functionalities. However, in harsh environments, these devices may compromise the overall system reliability since they are susceptible to Single Event Upsets (SEUs). These SEUs are events that can inadvertently change the configuration memory of the FPGA, thereby corrupting the function results and device outputs. These events are frequently caused by radiation, electromagnetic interferences and power fluctuations. Moreover, their intensity and frequency have been amplified by the trends of shrinking transistor dimensions and power reduction, hence requiring such errors to be handled through fault tolerance and error mitigation strategies.

Several mechanisms have been proposed to mitigate SEUs in SRAM-based FPGAs. The most common techniques exploit spatial/hardware redundancy [1] such as Triple Modular Redundancy (TMR) [2], [3] and Duplication With Compare (DWC) [4]. Besides TMR and DWC, blind scrubbing is often used to correct errors after the FPGA has been subjected to SEUs. This method of fault mitigation periodically rewrites the configuration frames of the FPGA, overwriting possible faulty bits caused by SEUs [5], [6]. Another technique of scrubbing stores Error Correction Codes (ECCs) corresponding to each frame either within the frame [7] or in an external memory [8]–[10], rewriting only those frames that have been corrupted by SEUs. ECCs are able to correct a number of errors in the frame by using additional redundant bits that help in the detection and correction of the errors. Common ECCs used in current works include parity codes, hamming codes and Reed Solomon codes. Such techniques overcome the need for a large external memory and have a good fault-tolerance against both single and burst SEUs.

Scrubbing-based on ECCs has been of particular interest to researchers, and a number of algorithms have been proposed using this technique. Examples of such algorithms include Matrix Codes [10], 2-D Hamming [8], interleaved and compressed hamming codes [9], and 3-D hamming schemes [11]. However, most of these proposed solutions are theoretical and do not present details on the implementation concerns. This lack of a good supporting architecture and design flow to enable the execution of these solutions motivate the need for a generic scrubbing architecture that is able to handle different algorithms and hence provide a flexible and convenient way to implement fault-tolerant designs on FPGAs. To the best of the authors knowledge, no other works have proposed a similar architecture to support various algorithms based on scrubbing.

The **key contributions** of this paper are:

- a generic scrubbing-based FPGA architecture and design flow for any error detection/correction algorithms based on ECC;
- an architecture that provides error monitoring and error correction;
- implementation and evaluation of three different algorithms demonstrating the feasibility and efficiency of the proposed architecture and design flow.

The rest of the paper is organised as follows. Section II presents the background concerning the FPGA reconfiguration mechanisms and the scrubbing mechanisms. Section III presents related works in the field of scrubbing-based hardware implementation architectures. In Section IV the proposed solution is described. Section V presents the design flow concerns. Section VI evaluates a case study and the presents the corresponding results. Finally, Section VII presents the conclusions.

II. Background

A brief introduction of the FPGA architecture and common scrubbing techniques is presented before the discussion of related works.

A. FPGA Configuration

FPGAs are configured using bitstreams, which contain the configuration data for each part of the FPGA such as the Block-RAMS (BRAMs), Look-Up Tables (LUTs), interconnections and Flip Flops (FFs). These individual blocks in the FPGA are accessible to the user for building a custom design. The FPGA can be seen as a collection of *frames*, in which each of these individual blocks are implemented. For example, the Xilinx Virtex-6 FPGA (XC6VLX240T-f1156) contains $28,464$ frames in total and each frame contains $2,592$ bits. Current FPGAs provide the ability to reconfigure at runtime both the structure and functionality of the implemented design. This is made possible by the reconfiguration ports in the FPGA. For example, the Virtex-6 series provides 3 different ports for reconfiguration: JTAG, SelectMAP and ICAP.

The Internal Configuration Access Port (ICAP) is an internal port that can be accessed within the FPGA for reconfiguration while the other 2 ports are external. The smallest region accessible for reconfiguration is a frame. Any frame in the FPGA can be reconfigured by addressing it via the Frame Address Register (FAR).

B. Scrubbing

Scrubbing is a very common technique to correct SEUs in FPGAs. This method uses an external memory to store the entire bitstream, called *golden copy*. The FPGA is then reconfigured at regular intervals using this golden copy to prevent the build up of errors. This technique, more specifically referred to as *blind scrubbing* [6], is very easy to implement. However, it is not efficient since it wastes a lot of FPGA resources in scrubbing parts of the FPGA which do not contain the user design. Moreover, it also incurs a large memory overhead due to the need for a golden copy.

C. Scrubbing based on Error Correction Codes (ECC)

In order to overcome the disadvantages posed by blind scrubbing, scrubbing techniques based on Error Correction Codes (ECCs), which do not incur a large memory overhead, have been proposed [8]–[11]. In such a technique, ECCs such as parity or hamming codes are frequently used to detect and correct errors in a frame. ECCs are computed for the user design at design time and stored in an extra memory. Such a memory is much smaller than the golden copy used for blind scrubbing owing to the smaller size of the ECCs. At runtime, the ECCs for each frame are computed and compared with the ones in the extra memory. If there is a mismatch in the ECCs, the errors are corrected and the corrected frame is written back into the FPGA. In recent years, numerous algorithms have been proposed to correct errors in the FPGA. All these techniques require a slightly different hardware architecture to implement them and efficiently correct the errors in the bitstream. This motivates the need for a generic scrubbing architecture that is able to handle different error detection/correction algorithms based on ECCs and hence provide a flexible and convenient way to implement fault tolerant designs on FPGAs.

TABLE I
Comparison of related works

Feature	Xilinx [5]	Straka [12]	Proposed work
Flexible architecture	No	Limited	Yes
Architecture overhead	None	TMR/DMR	Yes (ECC dependent)
Module size	Small	Big	Small
Handle burst errors	No	Yes (Limited)	Yes (ECC dependent)

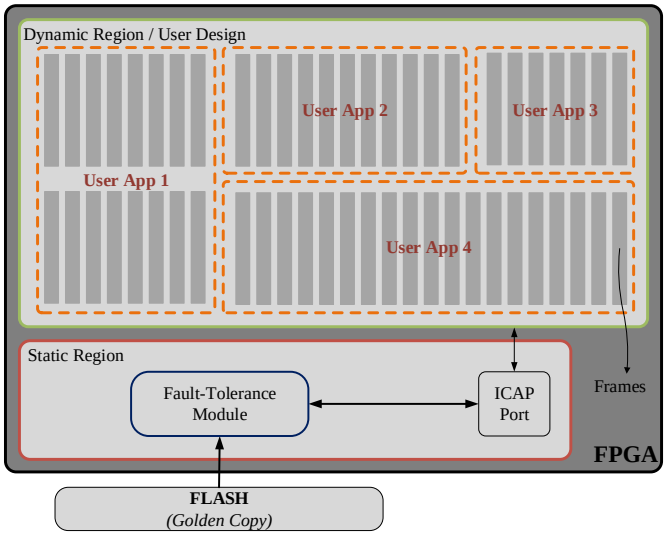

Fig. 1. Top level FPGA implementation perspective.

III. Related Works

There exist a number of works presenting scrubbing architectures to make the user design fault tolerant. However, they come with their own set of advantages and disadvantages. This section describes a few important works along with their pros and cons.

Xilinx provides a pre-verified solution to detect and correct errors in FPGAs [5]. This solution, called Soft Error Mitigation (SEM), is able to correct up to 1 error per frame in Virtex-6 FPGAs and is able to detect up to 2 errors per frame with a typical detection latency of 18 ms and an added correction

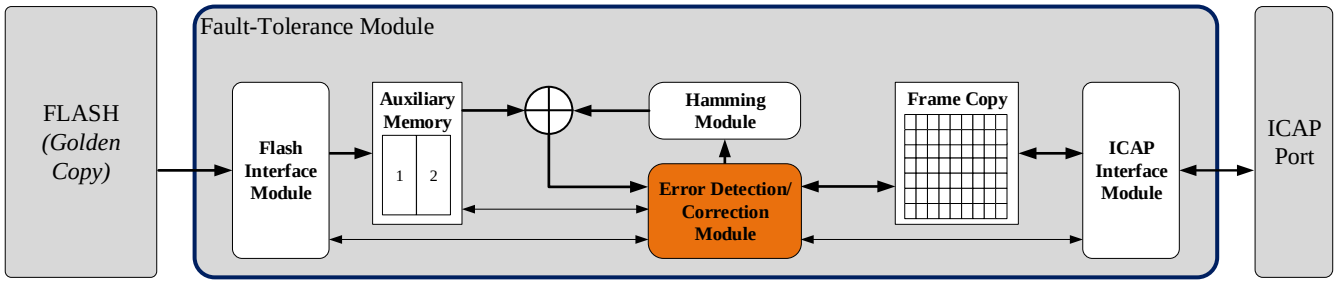

Fig. 2. Fault-Tolerance Module – implementation architecture.

latency varying from $20\mu s$ for uncorrectable errors to $660\mu s$ for correction by the "replace" technique. The advantages of this controller are that the solution comes pre-verified and is also integrated in the FPGA flow process. Hence, its implementation is straight forward with the user only having to interface the controller with the user design. However, the SEM controller can only detect up to 2 errors per frame and is hence not suitable for multi-bit upsets. Nowadays, multi-bit upsets are becoming an important issue, since they increase when technology scales down [13]. Moreover, multi-bit upsets are also stronger in the space environment. There, the hardened FPGAs give a suitable response to this problem. However, this kind of FPGAs is really costly, which is incompatible with a new category of budget space devices (like nano-satellites). In this sense, works and fault-tolerant architectures applied on COTS FPGAS that use more complex ECCs are gaining special importance [14]. On the contrary, the SEM controller does not support user-defined ECCs, thereby its flexibility and error correction is limited.

Straka et al. [15] propose four different fault-tolerant FPGA architectures coupled with Partial Dynamic Reconfiguration (PDR). This technique applies PDR through a hardware controller and hence performs well in terms of speed and area. In addition to the fault tolerant architecture, the authors also propose a fault simulation framework to inject FPGAs with SEUs. This framework is implemented externally to the FPGA and injects errors through the JTAG interface. Though the error correctability of such an architecture is quite good, it still incurs a large penalty in terms of the area and power overhead since the underlying fault masking principle still uses TMR or DWC with voters or checkers. Additionally, the architectures proposed are not flexible to other forms of error masking or correction techniques.

Various other works [16]–[18] have been proposed that are quite similar to the above mentioned works. However, these works lack proper implementation of the proposed architecture. As can be seen from Table I, current works do not provide a flexible architecture to implement different error detection/correction algorithms. Moreover, current works lack a simple way to detect and correct burst errors without expensive redundancy schemes. In this work, we aim to provide a flexible scrubbing architecture and a design flow with a robust implementation, which the user can use to implement various algorithms.

IV. Proposed Architecture

This section describes the features of the proposed generic scrubbing-based hardware architecture, called the Fault-Tolerance Module (FTM). This architecture allows custom error detection/correction algorithms, which use ECCs, to be implemented in a uniform design flow. Figure 1 describes the top level architecture of an FPGA, which is divided in two main regions — the *Dynamic Region* and the *Static Region*. The dynamic region is composed of several applications belonging to the user design. Through reconfiguration of the FPGA, user applications can be easily added or removed from the user design. The *Static Region* is the area that remains static during the runtime of the FPGA and is an application-independent region that is used to implement the logic of the partial dynamic reconfiguration. Due to this independence from the user design, it is an ideal region to place the FTM. The FTM has two main interfaces — the ICAP configuration port, which allows reading and writing frames from the user design, and the flash memory port, which allows reading from the flash memory. The flash memory contains the original ECCs, which were previously computed for all the frames of the user design as described in Figure 1.

The FTM consists of several sub-modules, with each one implementing a specific function. Figure 2 shows the overall FTM architecture, as well as the interconnection schematic among the different sub-modules. The subsequent subsections introduce these sub-modules and describe their function.

A. ICAP Interface Module (ICAPIM)

The ICAP Interface Module implements the suitable finite state machine to read and write individual frames to/from the user design. The reading process of a particular frame is triggered by the Error Detection/Correction Module (EDCM) and can be summarised by the steps shown in Figure 3–Top. The reading process is initialized by the synchronization step, which performs the necessary operations to synchronize the ICAP and set up the "read mode". Then, the address of the frame to be read has to be written to the ICAP frame address register and the reading operations can be performed. During this step, the ICAP Interface Module (ICAPIM) receives the frame data content word by word, through a 32-bit data bus.

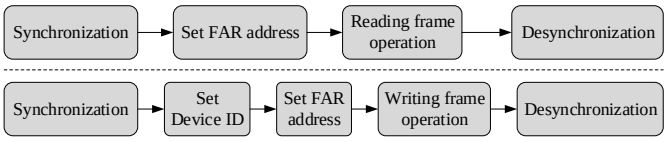

Fig. 3. **Top)** ICAP frame reading flow; **Bottom)** ICAP frame writing flow

For example, Virtex-6 frames are composed by 81 words [19] and Virtex-5 by 41 words [20]. Each frame word received from the ICAP port is then stored inside registers within the Frame Copy (FC). After the reading step, the ICAP desynchronization operations are executed.

The writing process is also triggered by the EDCM upon the detection and respective correction of the errors in the Frame Copy. The writing process follows a similar flow to the reading process, as described in Figure 3–Bottom [21], [22]. The synchronization step is slightly different, since the ICAP has to be set up in the "write mode". Moreover, another different is related with the device ID that has to be written in the respective ICAP register.

B. Frame Copy (FC)

The Frame Copy implements a set of registers to store all the words of a single frame, creating a 2D-bit matrix. By storing the frame in registers, the error detection/correction algorithm implemented in the EDCM can quickly access the frame content in different ways, such as matrix rows, matrix columns or even matrix diagonals. For instance, the algorithms proposed by 2-D Hamming [8] and P^2H/H^3 [11] require the access to the matrix columns, i.e., the concatenation of one bit of each register word. The implementation of this Frame Copy memory is a way to overcome an intrinsic COTS FPGA architectural limitation, since the COTS FPGA architecture does not provide a direct access to the contents of the frame as columns or diagonals through the ICAP Port.

C. Flash Interface Module (FIM)

The Flash Interface Module implements the hardware logic to interface with the flash memory. At design time, the computed ECCs of the user design frames are stored in the flash memory. During the runtime execution, the ECCs of each frame are read in the "burst mode". With this option, the flash reading process is faster, since flash device initialization steps are only required for each frame and not for each ECC read. The reading operation is triggered by the EDCM, which sends an enable read signal and a flash address to the Flash Interface Module. The flash address indicates the starting position where the ECCs of a given frame are stored. During the reading operation the ECCs are sent to the Auxiliary Memory, which stores them internally.

D. Auxiliary Memory (AM)

Typically, the flash memory operates at a different frequency than the ICAP. Therefore, in order to eliminate the need for synchronization, an Auxiliary Memory is inserted in the flash reading chain. This Auxiliary Memory has the capacity to store the ECCs of two frames from two different regions. Therefore, during the operation of error detection in one frame, using the respective ECCs stored in one of the regions of the Auxiliary Memory, the Flash Interface Module can parallely read the respective ECCs of the next frame to be processed. These ECCs are then stored in the other region of the Auxiliary Memory that is not in use.

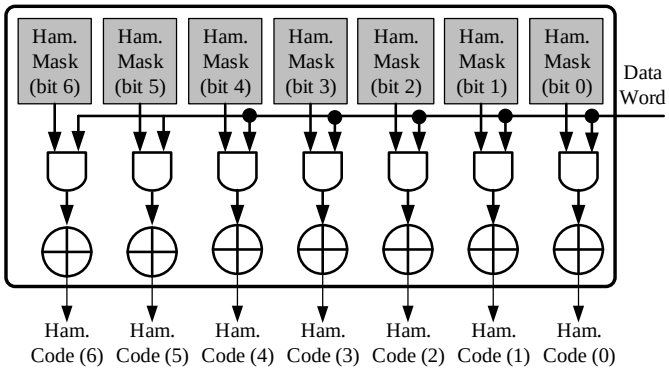

Fig. 4. Hamming Module – implementation architecture.

E. Hamming Module (HM)

The Hamming Module implements the necessary hardware to compute the hamming codes of a generic input word with a maximum length of $128 - 7$ bits. The 7 bits are the number of hamming bits that are embedded in a 128-bit word. Figure 4 describes the internal hardware logic. Each hamming bit is generated by XORing the significant input word bits defined by the corresponding mask for that hamming bit. The computed hamming code is then XORed with the original hamming code stored in the flash, to identify any error in the data word and its location.

F. Error Detection/Correction Module (EDCM)

The EDCM plays a central role since it triggers and controls all the operations in the architecture. Figure 5 presents the flowchart that describes the sequence of operations. When the execution starts, the ECCs of frame *n* are read from the flash and stored in one of the regions of the Auxiliary Memory. Then, frame *n* is read from the ICAP and stored in the Frame Copy. After these two operations, all the necessary data becomes available for the execution of the custom error detection/correction algorithm. First the error detection operation is executed. If any error is detected in frame *n*, the error correction operation is executed and frame *n*, which is placed in the Frame Copy memory, is written back to the user design through the ICAP Port. If frame *n* is free of errors the next frame on the user design is read through the ICAP Port. Note that during the reading of frame *n* from the ICAP, the ECCs of frame *n+1* are read from the flash device. With this process, the corresponding ECCs of frame *n* are available to be used by the error detection/correction algorithm when

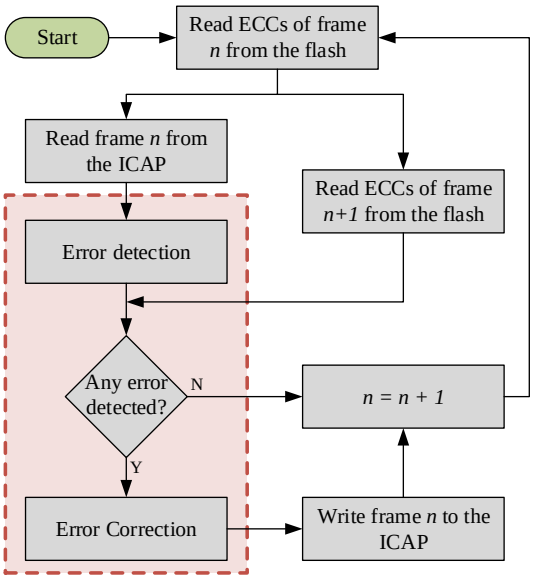

Fig. 5. Error Detection/Correction Module – operations flowchart.

it starts processing frame *n*. Therefore, any interruption on the algorithm execution is avoided, since the flash memory is typically slower than the ICAP.

The dashed line in Figure 5 identifies the operations performed by the custom error detection/correction algorithm. The area of influence of these operations is well defined, allowing an easy exchange of the algorithm in the proposed architecture. Any exchange of the used algorithm can be done transparently, i.e, without any interference or required modifications in the other architecture modules.

V. Design Flow

Figure 6 illustrates the overall design flow of the proposed generic scrubbing-based architecture. The steps in green highlight the traditional flow of the design implementation. The red and blue blocks are additional steps that have been added in the proposed technique.

The first step in the flow is the generation of the error correction module macro. This macro consists of the entire FTM including the error detection/correction algorithm specified by the user. As mentioned in Section IV-F, the architecture has been implemented in a way that is easy for the user to insert any algorithm in the EDCM. Once the error correction module macro has been generated, the macro is added to the user design and is synthesized and routed along with the user design. Once the bitstream is generated for the entire design, including the error correction module macro, ECC codes are computed and stored in the flash memory, which is looked up during the runtime of the FPGA. The FPGA is then finally configured with the generated bitstream. Once the user design starts to run on the FPGA, the EDCM is initialized and controls the error detection and correction processes as has been explained in Section IV-F.

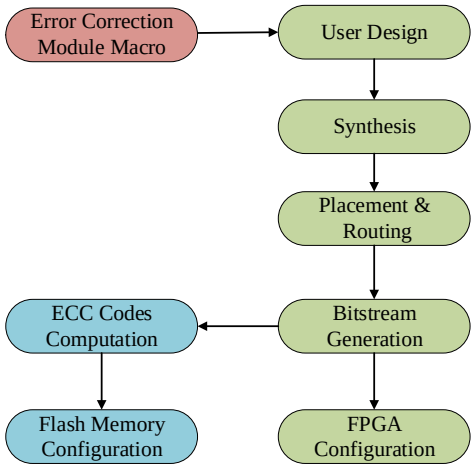

Fig. 6. Proposed solution – design flow.

VI. Case Study and Experimental Results

This section evaluates the proposed hardware architecture taking into account three error detection/correction algorithms that already have been published. The next subsection presents in detail these three algorithms.

A. Evaluated Error Detection/Correction Algorithms

Park *et al.* [8] propose a built-in 2-D Hamming Product Code ($2D - HPC$) scheme. This technique, using ECC-based scrubbing, is able to perform error correction by using hamming codes built from arranging the FPGA configuration frame in a 2-D array. Hamming codes are computed for each row and column of this 2-D array at design time. At runtime these rows and columns are iterated while computing their ECCs and comparing with the ones stored earlier. The algorithm iterates the rows and columns until all errors have been corrected or until a predetermined number of iterations is reached.

Venkataraman *et al.* [11] propose two ECC-based scrubbing schemes to detect and correct errors in FPGAs. The H^3 scheme applies SECDED hamming codes to a 2-D frame matrix in 3 different directions: rows, columns and diagonals. They claim that applying hamming code to diagonals improves fault tolerance for burst errors. Moreover, the P^2H scheme applies parity codes to the rows and columns, while applying hamming codes to the diagonal. The P^2H scheme has a lower ECC memory overhead than the H^3 scheme due to the use of parity codes for the frame rows and columns.

Table II summarizes the ECC memory overhead regarding the FPGA frame size and error correction percentage[1] for each of the schemes discussed. H^3 is ideal for applications that require a high level of reliability, without any concern about the resources used to achieve that level. P^2H is ideal for applications that have more concerns on the memory resources. Therefore, they are looking for a solution that uses

[1]20 errors injected per frame (32×32); H^3 and P^2H results shown for optimal schemes

TABLE II
RESULTS OF PREVIOUS WORKS ON ECC-BASED SCRUBBING

	ECC memory overhead	Error correction %
2D-HPC scheme [8]	40%	98%
H^3 scheme [11]	60%	100%
P^2H scheme [11]	25%	80%

a lower number of resources, but at the same time keeps the reliability levels high. Finally, the $2D - HPC$ scheme is a combination of the previous two.

B. Proposed Architecture Implementation Results

The proposed architecture has been implemented and verified on two different Xilinx boards (one with a Virtex-5 XUPV5-LX110T and the other with a Virtex-6 XC6VLX240T-f1156) with the three different error detection/correction algorithms described in the previous subsection. The entire architecture executes at 100MHz, the maximum frequency recommended by Xilinx for the ICAP configuration port. Table III presents the FPGA resources used by each module in the FTM (Figure 2). As can be observed, the proposed FTM is very dependent on the algorithms used to detect and correct the errors in the configuration frames. The EDCM that implements the algorithms is the module that uses most of the FPGA resources. Among the algorithms, the P^2H, being the most complex, is also the one that consumes more FPGA resources when compared to the others implemented in the same FPGA. Moreover, P^2H also has the lowest maximum working frequency. When we compare the FPGA resources consumed by EDCM using same algorithm and implemented in Virtex-5 and Virtex-6 FPGAs, the difference can be explained by the length of the Frame Copy. In Virtex-5 the frame length is 41 words, in contrast to the 81 words per frame in Virtex-6. Moreover, the logic to access the 81 registers in the Frame Copy is bigger and more complex when compared to the one that accesses 41 registers.

Table IV compares the proposed fault-tolerance architecture with the Xilinx SEM IP Core. The proposed solution uses more FPGA resources than the Xilinx solution for any algorithm implemented in the EDCM. This trend is expected, since the Xilinx solution is simpler than the implemented algorithms in the EDCM. However, the Xilinx solution can only correct one error per frame, since it generates the hamming codes for the entire frame. In contrast, the implemented algorithms, although more complex, present a much better error correction. For instance, they can correct up to 55 errors per frame 100% of the time (for the H^3 scheme).

C. ICAP Controller

Table V presents the measured time to read and write one FPGA configuration frame through the implemented ICAP controller. The Virtex-5 reading and writing times are faster than the Virtex-6 ones due to the difference in the size of the frames. Virtex-5 contains only 41 words per frame as compared to 81 words for Virtex-6. This leads to a larger time in reading and writing the frames for Virtex-6. Moreover, the reading of a frame takes more time than the writing of the frame due to the difference in the ICAP protocol for the two processes. Since the reading protocol has an additional synchronization steps before the frame can be read from the ICAP Port, the time taken to read the frame is longer than the time taken to write it.

TABLE IV
FPGA RESOURCES USED BY THE PROPOSED FTM

Mechanism	Slices Registers (%)		LUTs (%)	
	Virtex-5	Virtex-6	Virtex-5	Virtex-6
Xilinx SEM	1	1	1	1
FTM with $2D - HPC$	2	1	7	4
FTM with H^3	3	1	9	8
FTM with P^2H	3	1	15	15

TABLE V
ICAP – TIMING RESULTS.

Frame Operation	Virtex-5 (μs)	Virtex-6 (μs)
Reading	1.55	2.36
Writing	1.12	1.94

D. Error Detection/Correction Algorithms – Timing Results

In order to test and evaluate the implemented error detection/correction algorithms, two main operations were considered. The first one is the time to detect errors in one frame. This time includes the time to read one frame from the ICAP port and to execute the algorithm. The second one is the time to correct one error in a frame. This time includes the time to read a frame from the ICAP port, the time to detect and correct one error and finally the time taken to write back the corrected frame into the ICAP port. For all the hardware implementations, the clock frequency was set to 100MHz. Table VI presents the obtained execution times. As expected, the algorithms' execution times on Virtex-6 are higher than those of the implementations on Virtex-5, since the frame size in Virtex-6 (81 words) is bigger than the one in Virtex-5 (41 words). Regarding the implementations on the same FPGA, the obtained timing results are also according to expectations. The $2D-HPC$ and P^2H have almost the same performance. The former verifies the existence of errors on the rows and columns of the frame matrix. The latter, for a small number of errors, only checks for errors on one diagonal of the matrix frame. H^3 is the slowest, since it has to find errors on rows, columns and diagonal of the frame matrix.

VII. CONCLUSION

With the increased importance of embedded systems and proliferation of COTS FPGAs in both academic and commercial sectors, the number of SEUs affecting such devices have also been on the rise. This paper proposes a generic architecture for such COTS FPGAs using scrubbing-based

TABLE III
FPGA RESOURCES USED BY THE PROPOSED FTM

Module	Slices		LUTs		BRAMs		FFs		Max. Frequency (MHz)	
	Virtex-5	Virtex-6	Virtex-5	Virtex-6	Virtex-5	Virtex-6	Virtex-5	Virtex-6	Virtex-5	Virtex-6
ICAPIM	125	104	378	273	0	0	246	246	217.85	258.09
HM	13	9	19	19	0	0	6	7	245.76	411.69
AM	0	0	0	0	1	1	0	0	-	-
FIM	10	15	20	25	0	0	55	34	338.61	508.38
EDCM+FC with $2D-HPC$	1818	2916	4490	8240	0	0	1519	2876	147.92	168.86
EDCM+FC with H^3	2155	4235	5930	15001	0	0	1552	2931	145.61	167.88
EDCM+FC with P^2H	4209	7758	9667	23301	0	0	1860	3338	144.64	164.59

TABLE VI
ERROR DETECTION/CORRECTION ALGORITHMS – TIMING RESULTS.

Algorithm	Operation	Virtex-5 (μs)	Virtex-6 (μs)
$2D-HPC$	Detection	2.30	3.51
	Correction	4.17	6.60
H^3	Detection	3.82	4.64
	Correction	5.61	8.80
P^2H	Detection	2.31	3.52
	Correction	4.19	6.62

custom error correction algorithms. The architecture enables the implementation of any algorithm that is based on scrubbing and provides a simple design flow for the application synthesis. Moreover, the architecture also supports error monitoring and error correction. Such a design allows the user to choose the error detection and correction algorithms suitable for the application and implement them easily using the FTM macro.

The architecture has been implemented and verified on two different Xilinx boards (Virtex-5 XUPV5-LX110T and Virtex-6 XC6VLX240T-f1156) with 3 different error detection/correction algorithms proposed recently by other authors. The implemented architecture occupies under 15% of the overall FPGA size (for a Virtex-6 board) while taking care of the error detection and correction process through reading and writing frames from the FPGA ICAP module. A thorough breakdown of the individual modules and their resources used have also been presented.

Plans for the future work include releasing an open-source tool to help integrate the proposed architecture into the user's design. Such a tool would enable researchers across the world to compare error detection/correction algorithms based on scrubbing on a common architecture and evaluate their performance. Moreover, further optimization on the proposed architecture in terms of area and power overhead will also be carried out in extension to the current work.

REFERENCES

[1] I. Koren and C. Krishna, *Fault-tolerant systems*. Morgan Kaufmann, 2007.

[2] B. Pratt, M. Caffrey, P. Graham, K. Morgan, and M. Wirthlin, "Improving FPGA Design Robustness with Partial TMR," in *44th Annual IEEE International Reliability Physics Symposium Proceedings*, 2006.

[3] C. Bolchini, D. Quarta, and M. D. Santambrogio, "SEU mitigation for SRAM-based FPGAs through dynamic partial reconfiguration," in *Proceedings of the 17th ACM Great Lakes symposium on VLSI*, 2007.

[4] A. Sarkar, F. Mueller, H. Ramaprasad, and S. Mohan, "Push-assisted migration of real-time tasks in multi-core procs," *SIGPLAN Not.*, 2009.

[5] C. Carmichael, M. Caffrey, and A. Salazar, "Correcting Single-Event Upsets Through Virtex Partial Configuration," Xilinx, Tech. Rep., 2000.

[6] J. Heiner, B. Sellers, M. Wirthlin, and J. Kalb, "FPGA partial reconfiguration via configuration scrubbing," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, 2009.

[7] K. Chapman, "SEU Strategies for Virtex-5 Devices," Xilinx Application Note, 2010.

[8] S. P. Park, D. Lee, and K. Roy, "Soft-error-resilient FPGAs using built-in 2-D Hamming product code," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 20, pp. 248–256, 2012.

[9] M. Lanuzza, P. Zicari, F. Frustaci, S. Perri, and P. Corsonello, "A self-hosting configuration management system to mitigate the impact of Radiation-Induced Multi-Bit Upsets in SRAM-based FPGAs," in *IEEE International Symposium on Industrial Electronics*, 2010.

[10] C. Argyrides, D. K. Pradhan, and T. Kocak, "Matrix codes for reliable and cost efficient memory chips," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2011.

[11] S. Venkataraman, R. Santos, S. Maheshwari, and A. Kumar, "Multi-directional error correction schemes for SRAM-based FPGAs," in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, 2014.

[12] M. Straka, L. Miculka, J. Kastil, and Z. Kotasek, "Test platform for fault tolerant systems design properties verification," in *Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2012 IEEE 15th International Symposium on*, 2012.

[13] E. Ibe, H. Taniguchi, Y. Yahagi, K. Shimbo, and T. Toba, "Impact of Scaling on Neutron-Induced Soft Error in SRAMs From a 250 nm to a 22 nm Design Rule," *Electron Devices, IEEE Transactions on*, 2010.

[14] P. M. B. Rao, M. Ebrahimi, R. Seyyedi, and M. B. Tahoori, "Protecting SRAM-based FPGAs Against Multiple Bit Upsets Using Erasure Codes," in *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC '14, 2014.

[15] M. Straka, J. Kastil, Z. Kotasek, and L. Miculka, "Fault tolerant system design and SEU injection based testing," *Microprocessors and Microsystems*, vol. 37, pp. 155–173, 2013.

[16] Z. Ghaderi, S. Miremadi, H. Asadi, and M. Fazeli, "HAFTA: Highly Available Fault-Tolerant Architecture to Protect SRAM-Based Reconfigurable Devices Against Multiple Bit Upsets," *Device and Materials Reliability, IEEE Transactions on*, vol. 13, pp. 203–212, 2013.

[17] G. L. Nazar, L. P. Santos, and L. Carro, "Scrubbing Unit Repositioning for Fast Error Repair in FPGAs," in *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2013.

[18] A. Harding and M. Wirthlin, "Improving the Reliability of Xilinx 7 Series FPGAs through Configuration Scrubbing," 2014.

[19] *Virtex-6 FPGA Configuration User Guide*, UG360 (v3.8), Xilinx Inc., 2014.

[20] *Virtex-5 FPGA Configuration User Guide*, UG191 (v3.2), Xilinx Inc., 2008.

[21] *ML505 Evaluation Platform User Guide*, UG347 (v3.1.2), Xilinx Inc., 2011.

[22] *Virtex-6 FPGA ML605 Hardware User Guide*, UG534 (v1.8), Xilinx Inc., 2012.