Data content scrubbing approach for SRAM based FPGA designs

J. Gomez-Cornejo*, I. Villalta†, I. Aranzabal*, I. Lopez*, and A. Zuloaga†

*Dept. of Electrical Engineering. University of the Basque Country UPV/EHU. Bilbao, Spain
†Dept. of Electronic Technology. University of the Basque Country UPV/EHU. Bilbao, Spain

2022 IEEE 31st International Symposium on Industrial Electronics (ISIE) | 978-1-6654-8240-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/ISIE51582.2022.9831467

***Abstract*—The configuration memory of SRAM-based FPGAs can be susceptible to induced faults potentially causing errors that may impact devices' functionality (depending on the criticality of the affected bit). The accumulation of those errors increase the probability of malfunction. The scrubbing is a hardening technique utilized to refresh the configuration memory of the FPGA, which defines the functionality of the device and can store user data content. While, scrubbing of the configuration memory of FPGAs is a well established strategy, the scrubbing of the data content has not been sufficiently addressed. Due to this, the present work proposes a scrubbing approach to clean errors from the data memories implemented as BRAMs or distributed memories based, and physically validated, on the ZYNQ from Xilinx.**

***Index Terms*—FPGA, fault, memory, scrubbing, bitstream.**

I. Introduction

An ideal electronic design should always work without malfunction. Nevertheless, in a real scenario electronic systems are exposed to certain risks of failure. A dependable design should try both, to avoid the impact of errors and to repair the produced ones. Related with those ideas, two concepts have to be remarked: availability and reliability. While availability is determined by the mean time to repair after a failure, reliability is determined by the design itself, the platform utilized and the operation-environment. It is important to obtain a proper balance of both concepts. For instance, a design with poor reliability level but with a fast repairing method most probably will meet its goals in presence of faults. On the other hand, a very reliable design that needs complex and time demanding repairing techniques perhaps will not achieve its goals after an error. Reliability is a valuable characteristic, but availability is what the final user is going to experience. In order to provide satisfactory reliability and availability levels for FPGA (Field Programmable Gate Array) designs, an adequate fault tolerance level must be guaranteed. Due to this, increasing fault tolerance level of SRAM FPGA designs is one of the main objectives of this research work. Different strategies can be adopted to increase the fault tolerance level of FPGA designs in order to avoid possible negative consequences of induced faults.

One of the most remarkable method to account for and overcome potential faults is the so-called bitstream scrubbing [1]–[3] which takes advantage of the partial reconfiguration capability of FPGAs. The bitstream is a configuration file that defines the functionality of the FPGA. The bitstream scrubbing is performed by rewriting a known bitstream (named *golden* bitstream) with a correct configuration. The scrubbing cleans upsets from the configuration memory and prevents their accumulation. Thus, significantly reduces the probability of various of those errors being present at the same time.

Two strategies can be adopted to decide when performing the scrubbing. The first consists in periodically reconfiguring the device by utilizing the *golden* bitstream. This strategy is known as *blind scrubbing* [4]. The second strategy is based on carrying out the scrubbing after a fault detection [5], like in [6] where the *lazy scrubbing* approach is proposed. This method was applied to a hardware redundancy based scheme in which, the configuration bitstream was read from all replicated modules analyzing data and offsets to repair the faulty module. As this work stated, the *lazy scrubbing* demands less power and resource overhead and produces less single point of failures than the traditional scrubbing. There is also the possibility of combining both strategies. In this way, a periodical scrubbing can be scheduled to be performed in convenient stages maintaining the possibility of triggering and emergency scrubbing after a fault detection.

The scrubbing requires some type of control mechanism to communicate with the reconfiguration interface in order to load the *golden* bitstream. Due to the complexity of this task the prevalent solution is to utilize a processor as scrubbing controller. Another fundamental requirement is a memory module to store the *golden* bitstream. Following these ideas three predominant scenarios [7] can be introduced:

1) The on-chip scrubbing, shown in Figure 1, is the most compact solution, since no external element is required. A soft-core based scrubbing controller implemented in the target FPGA [8] performs the bitstream rewriting process by reading the *golden* bitstream from a BRAM (Block RAM) memory block and downloading it to itself through the internal configuration port (i.e., ICAP or PCAP). The main benefits of this auto-configuration process are simplicity and self-sufficiency. Nevertheless, it presents a critical drawback: since the control logic and the gold bitstream storage are implemented in the FPGA fabric, they are susceptible to SEEs (Single Event Effects), affecting the reliability level. In [9], a fault tolerant ICAP scrubber was presented to overcome this limitation. It consist in triplicating the internal ICAP circuit. However, it does not avoid the presence of single point of failures.

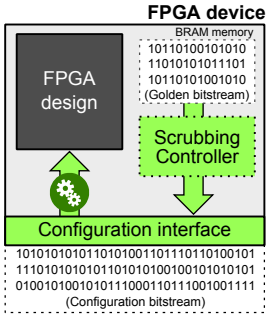


Fig. 1. On-chip scrubbing.

2) Another alternative is to utilize an external device to implement the scrubbing controller and the bitstream storage memory [4], [10]. In this case, the scrubbing controller performs the scrubbing through the external configuration port (i.e., JTAG or SelectMap). Figure 2 presents this scheme. The main advantage of this approach is the higher fault tolerance level, especially when the external device presents high reliability. As happens in [11] where the scrubbing controller is implemented in an external anti-fuse FPGA and the *golden* bitstream is stored in a hardened memory. The main drawback of this alternative is the complexity, since such a design requires higher power consumption (two devices to be feed), more physical space and additional hardware (communication buses, conditioners, etc.). Hence, this alternative demands a bigger design effort and increases the design's size, which is likely to increase the costs.

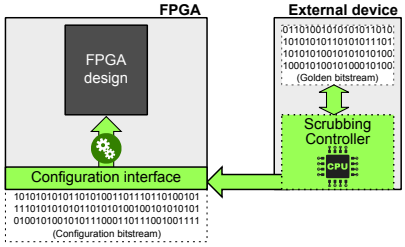


Fig. 2. Scrubbing with an external device.

3) Finally, Figure 3 shows the third alternative, the SoC (System-on-Chip) scrubbing based on the utilization of an FPGA device provided with an in-built hard-core processor, such as, the Zynq by Xilinx. In this case, the processor performs the scrubbing process utilizing the internal configuration port, while the *golden* bitstream is stored in an SoC's memory block, like a DDR module. This is a medium-cost compact solution, because an SoC device is likely to be more expensive than a simple FPGA but cheaper than implementing a two devices based system. It also presents an adequate reliability due to the high fault tolerance level of hard processors. The main drawback of this approach is that makes use of a valuable resource of the SoC, due to the fact that an SoC commonly has only one or two hard processors. Nevertheless, the utilization of this approach only demands the processor while the scrubbing process. This means that during the time between scrubbings the hard processor is suitable for other tasks. Hence, lower scrubbing frequencies mean higher availability of the hard processor.

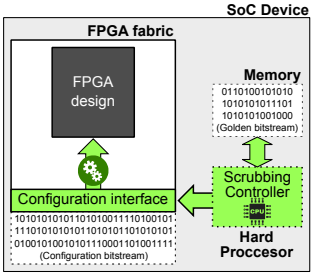


Fig. 3. Scrubbing with a hard processor in a SoC device.

Xilinx also provides with the possibility of taking advantage of the ECC (Error Correction Code) protection of the bitstream by utilizing the so-called *Soft Error Mitigation (SEM) Core* [12] or the *SEU controller macro* [13]. They check the ECC for errors in the configuration memory, repairing it when it is necessary. In [14], a similar strategy was presented.

The main weakness of the scrubbing resides in its time demand. Reconfiguration is a relatively slow process due to the limitations of configuration interfaces. In addition, the more data is to be rewritten, the more time is required due to the bigger size of the bitstream. Different solutions can be adopted to alleviate this handicap. The most straightforward one, as presented in [1], [15], is to use the placing constraints to locate the utilized resources in near placements. Due to this, the size of the partial bitstream is likely to be smaller. Nevertheless, placement constraints limit the freedom of the implementation software that can lead to worst results in terms of resource usage an performance. Other researches propose compressing the bitstream [16] to shrink it and reduce the duration of the partial reconfiguration process. Nevertheless, these methods are relatively complex since they require a compression mechanism, and they are also time demanding.

Considering that scrubbing is not able to repair certain elements of the FPGA (digital clocks managers, power on resets, selectMAP interfaces, etc.) it is not a definitive solution. In addition, considering the time demand on the reconfiguration operation, even in scenarios where a periodical scrubbing is a suitable alternative, the frequency of the scrubbing process is usually lower than application's, and hence the scrubbing by itself is not generally sufficient. Due to these factors, the majority of approaches utilize the scrubbing in combination with other hardening techniques, like hardware redundancy based approaches [11], [16]–[24].

Bearing in mind that the bitstream controls both, functionality (configuration and interconnection of logic resources) and user data content (BRAM, registers, etc.), two types of scrubbing can be defined: configuration and user data content scrubbing. Unless it is changed through reconfiguration to vary its functionality, configuration content is mostly a static

information. On the contrary, data content can be constantly updated during operation. By virtue of this fact, while the configuration is valid for almost all cases, the user data content scrubbing is suitable for certain cases where this data content remains unchanged for large periods, such as soft-core processors' program memories. In situations where the memory content changes runtime (i.e. data memories or registers of soft-core processors), performing the scrubbing of this information is not usually practical because of the performance penalty introduced by this process.

The most straightforward approach to scrub data content is to carry out a complete scrubbing by rewriting a bitstream with both, configuration and data content information. This scenario, means bringing back the system to the initial state of operation, which is a handicap. In additions, requires a power-cycling, which needs to stop the application. Hence, it is not a valid solution for applications that need to be continuously active. For this reason, the common practice is to perform only a configuration scrubbing repairing the static portion of the configuration memory that controls the logic of the FPGA, while the user data is hardened with a redundancy based scheme (hardware, data, software or time redundancy) able to mask errors [25], [26].

Although masking errors can be an interesting alternative, it is not reliable against the accumulation of errors. This situation is particularly problematic for the case of static memories, like program memories, where the possibility of accumulation of errors is higher. Due to this, certain studies have targeted user data memory scrubbing, like [27], where a user memory scrubbing approach to clean errors in memories was proposed. This user memory scrubbing hinges on the addition of an finite state machine based module to the design. Similar approaches were presented in [9], [28]. These methods increase resource overhead and need to use a second memory port. Since Xilinx FPGA BRAMs can only be implemented as single or dual-port memories, if a BRAM memory block is already being used as a dual-port memory this approach not be feasible. [3] describes the memory coherence problem. This issue appears when the bitstream gathers user data updated by system-level processes between writeback and readback functions. The presented work addressed this problem by using the so called *dirty-bit* technique, representing a step forward in this subject. Nevertheless, this method has a negative effect in the performance.

Since a lack of solutions to perform user data scrubbing has been identified, this work proposes an approach to circumvent the issues of the introduced methods. In this way, the proposed approach is able to carry out this process in a straightforward fashion based on the ZYNQ device.

The remainder of this document is structured as follows. Section 2 surveys the proposed approaches to perform user data scrubbing. Next, Section 3 presents the validation methodology and discuses the results. Finally, Section 5 outlines the conclusions and future work.

II. User data Content Scrubbing Approach

User data content can be stored by utilizing BRAM (Block RAM) or using distributed general-purpose fabric logic to implement distributed memories [29]. Choosing the best type of memory depends on each application and is a designing decision to be made by designers or the logic synthesis software of the FPGA. While distributed memories are mostly used to implement reduced memory structures with fast access, the utilization of BRAMs is mainly related with large memory structures that provide a sequential data access. Due to this, the developed approach to perform the data content scrubbing is divided in two techniques: data scrubbing of BRAM memories and data scrubbing of distributed memories.

A. Data scrubbing of BRAM memories

The developed approach to carry out the user data scrubbing of memories implemented in BRAMs is a SoC scrubbing method based on the methodology presented in [30]. It utilizes its implementation scheme shown in Figure 4. Since this approach identifies the location of data and its distribution through the bitstream, it makes feasible to both read and write the user data content stored in the bitstream. Depending on the number and placement of the BRAMs utilized by the remaining memory modules of the design, the most suitable alternative has to be chosen. This is because it has to be determined if the actual content of the remaining BRAMs has to be preserved or not. Thus, in cases where a BRAM column only contains memory blocks to be scrubbed, this process can be directly done. On the other hand, on implementations in which the same BRAM column contains both, memories to be scrubbed (i.e. program memories os soft-core processors) and memories to not be scrubbed (i.e. data memories of soft-core processors), the methodology to copy BRAM memories without overwriting has to be adopted in order to preserve the required information. Bearing in mind that this options demands higher processing effort, and consequently more time, it is advisable to avoid its utilization, when it is possible.

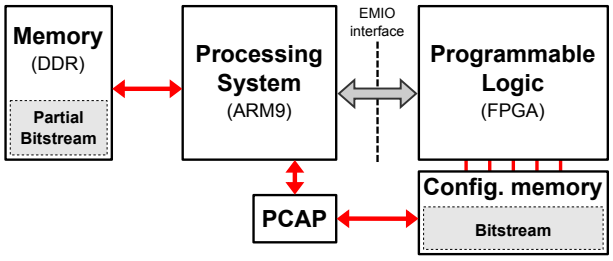
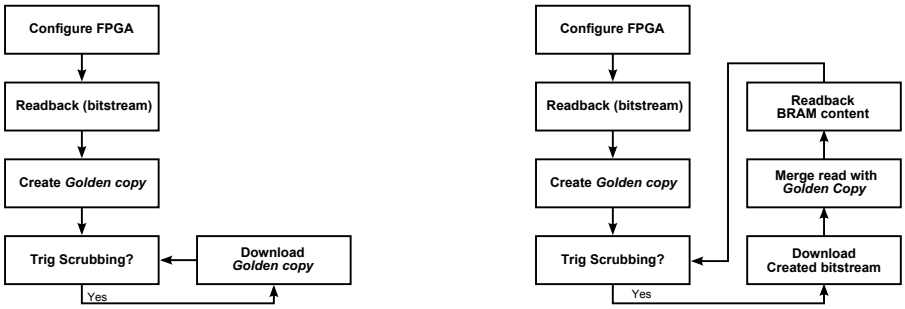


Fig. 4. Block diagram of the implementation scheme.

Considering that depending on the design, a BRAM column could be overwritten or not, two design flows are proposed to perform data content scrubbing. As it can be seen in Figure 5, both flows share the first four steps, which are used to obtain the *golden* bitstream. The first step is to configure the FPGA by downloading the bitstream. In a next step, a readback process is performed through the PCAP interface.



a. Scrubbing with overwriting. b. Scrubbing without overwriting.

Fig. 5. Flow charts of scrubbing BRAM based user data content

Thanks to this, the data obtained is properly structured and can be utilized to create the partial *golden* copy bitstream file in a straightforward fashion. These first steps could be substituted by extracting the data from the bitstream file. However, due to the complex data distribution in the bitstream file this process would require a high processing effort. In addition, each design requires specific processing due to the different BRAM locations. Once the *golden* copy is obtained and stored in an external memory (i.e. a DDR) the scrubbing process can be triggered. In the case that the remaining BRAMs of the BRAM column can be overwritten, the *golden* copy can be directly downloaded. But in cases where they cannot be overwritten, a readback operation has to be performed in order to store the actual data content and to be able of merging it with the *golden* bitstream. Depending on the design this process may require to stop the clock signal in order to assure stable information.

One limitation of this approach is the scrubbing frequency, since the time demanded by the bitstream download process affects to the time between consecutive scrubbings.

B. Data scrubbing of distributed memories.

As has been introduced data content scrubbing is mainly advisable for static memories, like program memories, which usually are implemented by utilizing BRAMs. For the rest of memory elements (data memory, registers, etc.) data scrubbing usually is not practical because the content is constantly updated during operation. However, even with those elements, in certain cases, this method could be an interesting alternative to initialize specific memory positions or registers. For instance, this could help to avoid the need of initialization instructions after resetting the system. For this reason, this work proposes an approach to perform data scrubbing in distributed memories.

Unlike what occurs with BRAM memories, in the case of distributed memories, its data content it is not directly included in the bitstream. However, the bitstream stores the initialization values of flip-flops with `INIT0/INIT1` initialization options [31], and those values can be updated performing a context capture with `CAPTUREE` primitive. The bitstream also includes SRVAL values that can be controlled by the local `SR` control signal. Hence, a capture of the state of registers can be carried out by using the `GCAPTURE` primitive, obtaining a *golden* copy of the data content of registers. To perform the scrubbing process, the `GSR` signal of the `STARTUPE2` primitive can be used, since it loads the `GRESTORE` command which triggers the initialization of the data content stored in `INIT0/INIT1` of the partial bitstream. This approach also follows the previously introduced implementation scheme from Figure 4. Both, capturing and scrubbing processes, can be triggered by the ARM processor by using the EMIO interface or by a dedicated control module implemented with the fabric logic. The ARM alternative is slower but it is way more reliable against induced faults than the module implemented with the fabric logic, which is more susceptible to induced faults. On the other hand, the DDR memory can be utilized to store the captured *golden* copy in order to avoid its corruption due to induced faults.

In this case, the approach is a very fast solution, however, it can require certain time depending on the design (a few clock cycles). The `GSR` is an asynchronous signal that does not require general-purpose routing. Bearing in mind that release and skew processes are done asynchronously, to avoid possible metastable events related with the different releasing of the flip-flops, it is advisable stopping the clock and waiting until the `GSR` spreads across the elements. This time requirement is a design dependant parameter. For instance, clustering tightly the registers minimizes the path length which also reduces the required time.

A limitation of this solution is that both, stores and loads the content of the entire device, which in some case will affect to undesired memories. A solution to this problem is based on the use of partial bitstreams. This is because if a partial bitstream is generated using `RESET_AFTER_RECONFIG=TRUE` [32] property, only the content of this particular partial bitstream will be affected by the effect of the `GRESTORE` command, while the rest will remain unaffected [33].

III. Physical Validation

The experimental setup of this research has been implemented in a ZedBoard by AVNET, which includes a Xilinx Zynq-7000 AP SoC XC7Z020-CLG484 device that contains tightly coupled 7-series programmable logic and dual-core ARM Cortex-A9 processing-system.

The approaches have been validated by running an application based on the soft-core processors presented in [34] performing a RS232 serial transmission based application. The utilized design is a small soft-core processor that makes use of a BRAM tile to implement program and data memories and distributed logic to implement internal registers and different modules. The serial transmission application sends a sequence of ASCII characters stored in the program memory thanks to a finite state machine based program. The performed experiments have provided the insight into correctness of the results obtained in a real world scenario.

In a first step, the *golden* copy of both, BRAMs and distributed memories content has been obtained. This required to previously capture the data content of registers by triggering the capture signal of the `GCAPTURE` primitive. After that, all the required data has been properly stored in the bitstream making it available performing the readback of the bitstream. In a next step, the *golden copy* has been stored in the DDR memory.

Next, in order to be able to generate several corrupted versions of the golden copy different data-related bits have been flipped. The corrupted partial bitstreams have been utilized to perform several error injection tests by reconfiguring the device with such corrupted partial bitstreams. After, those errors have been corrected by performing the scrubbing with the *golden* copy. Both alternatives of scrubbing for BRAM memories, with and without overwriting have been tested. Thanks to the performed tests it has been proved that this approach successfully performs a scrubbing of the data content of BRAMs and the distributed memories.

The results shown in Table I have been obtained using the *Flow_PerfOptimized_high* strategy in the synthesis and *Performance_ExplorePostRoutePhysOpt* in the implementation from Vivado. The first column of the table describes de resource analysed. The second column contains the results for the implementation of the serial transmission application that consist of the soft-core processor and the logic necessary. Finally, the third column contains the results of the implementation for the previous original design with the addition of the logic, hard-core processor and primitives necessary to perform the proposed user data scrubbing approach. The numbers between brackets indicate the utilization percentage for this specific device. As can be observed the proposed approach does not suppose a increase of logic resources overhead. Since the difference between with and without overwriting BRAM content resides in the processing of the bitstream, the resource utilization in both cases is identical. However, due to the use of the ARM hard-core processor and both, `STARTUPE2` and `CAPTUREE2` primitives, there is an increase of the power consumptions when applying the approach. In cases where the ARM hard processor is already utilized by the design itself, this aspect would be negligible.

In addition, as expected, the measurement of the time requirements of the approach has demonstrated that its main drawback is demanded time. Which on the other hand, is a common disadvantage of scrubbing approaches. As shown in table II, in the case of scrubbing with overwriting of BRAMs the required time is 2.5 ms. The case of scrubbing without overwriting the content of the BRAM column requires 11.2 ms. Results also show that scrubbing of registers' content is a very fast process, in this case the time demand of 0.4 µs is related with a waiting process programmed to ensure the spreading of the `GSR` signal across the path.

TABLE I
IMPLEMENTATION RESULTS SUMMARY
(@100MHz).

Resource	Original design	Design with scrubbing
Slice LUTs	229 (0.43%)	229 (0.43%)
Slice Registers	70 (0.07%)	70 (0.07%)
F7 Muxes	17 (0.06%)	17 (0.06%)
F8 Muxes	7 (0.05%)	7 (0.05%)
Block RAM Tile	1 (0.71%)	1 (0.71%)
CAPTUREE2	0 (0%)	1 (100%)
STARTUPE2	0 (0%)	1 (100%)
Dynamic p. (W)	0.115	1.638
Static p. (W)	0.125	0.161

TABLE II
TIMING RESULTS SUMMARY
(@100MHz).

Type of scrubbing	Time
BRAM (with overwriting)	2.5 ms
BRAM (without overwriting)	11.2 ms
Register	0.4 µs

IV. CONCLUSIONS AND FUTURE WORK

In contrast to what [27] stated, the present approach makes it feasible to perform a user memory scrubbing in BRAM and distributed memories by using the bitstream information. Just as it occurs with the configuration scrubbing, neither fabric logic resource overhead nor performance penalty of the implementation are incurred by the design. In addition, thanks to this proposed approach, the previously described memory coherence problem from [3] can be solved. Besides, in designs where the BRAM column can be overwritten the configuration scrubbing can performed in runtime without stopping the system, thus, without adding any performance penalty. All proposed approaches have been physically validated in an FPGA device running a real case applications, obtaining successful results.

A drawback of the proposed approach is the time demand, specially in the case of scrubbing the content of BRAM memories without overwriting the content of the rest of BRAMs of the column. This is mostly because of speed of the reconfiguration port and the amount the data to be processed. However, another reason is the use of Xilinx functions to perform those processes. Hence a possible line of study can be to optimize the programs used to couple more tightly with the specific application, for instance, by reading more than one frame at once.

Another remarkable line of research to give continuity to this work is to adapt and apply the ideas proposed in this work to other FPGA devices form other vendors.

ACKNOWLEDGMENTS

This work has been supported in part by the Government of the Basque Country within the fund for research groups of the Basque University system IT978-1 and in part by the Ministerio de Economía y Competitividad of Spain within the project TEC2017-84011-R and FEDER funds.

REFERENCES

[1] A. Sari and M. Psarakis, "Scrubbing-based SEU mitigation approach for systems-on-programmable-chips," in *International Conference on Field-Programmable Technology (FPT)*, 2011, pp. 1 – 8.

[2] J. Heiner, B. Sellers, M. Wirthlin, and J. Kalb, "FPGA partial reconfiguration via configuration scrubbing," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2009, pp. 99 – 104.

[3] W. J. Huang and E. J. McCluskey, "A memory coherence technique for online transient error recovery of fpga configurations," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, 2001, pp. 183 – 192.

[4] M. Berg, C. Poivey, D. Petrick, D. Espinosa, A. Lesea, K. LaBel, M. Friendlich, H. Kim, and A. Phan, "Effectiveness of internal versus external seu scrubbing mitigation strategies in a Xilinx FPGA: Design, test, and analysis," *IEEE Transactions on Nuclear Science*, pp. 2259 – 2266, 2008.

[5] N. Imran, R. A. Ashraf, and R. DeMara, "On-demand fault scrubbing using adaptive modular redundancy," in *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2013, p. 1.

[6] M. Garvie and A. Thompson, "Scrubbing away transients and jiggling around the permanent: Long survival of FPGA systems through evolutionary self-repair," in *IEEE International On-Line Testing Symposium*, 2004, pp. 155 – 160.

[7] U. Legat, A. Biasizzo, and F. Novak, "SEU recovery mechanism for SRAM-based FPGAs," *IEEE Transactions on Nuclear Science*, vol. 59, no. 5, pp. 2562 – 2571, 2012.

[8] C. Carmichael and C. Wei Tseng, "Correcting single-event upsets in Virtex-4 FPGA configuration memory," Xilinx Documentation, http://www.xilinx.com, Xilinx Corp., Tech. Rep., 2009, xAPP1088 (v1.0).

[9] J. Heiner, N. Collins, and M. Wirthlin, "Fault tolerant ICAP controller for high-reliable internal scrubbing," in *IEEE Aerospace Conference*, 2008, pp. 1 – 10.

[10] M. Kumar, D. Digdarsini, N. Misra, and T. Ram, "SEU mitigation of rad-tolerant Xilinx FPGA using external scrubbing for geostationary mission," in *India Conference (INDICON), 2016 IEEE Annual*. IEEE, 2016, pp. 1 – 6.

[11] M. Reorda, M. Violante, C. Meinhardt, and R. Reis, "An on-board data-handling computer for deep-space exploration built using commercial-off-the-shelf SRAM-based FPGAs," in *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2009, pp. 254 – 262.

[12] Xilinx Corp., "Soft error mitigation controller. PG036 (v4.1)," Xilinx Documentation, http://www.xilinx.com, 2014.

[13] K. Chapman, "SEU strategies for Virtex-5," in *Application Note: Virtex-5 Family*, 2009.

[14] G. Vera, S. Ardalan, X. Yao, and K. Avery, "Fast local scrubbing for field-programmable gate array's configuration memory," *Journal of Aerospace Information Systems*, pp. 144 – 153, 2013.

[15] G. Asadi and M. Tahoori, "Soft error mitigation for SRAM-based FPGAs," in *IEEE VLSI Test Symposium*, 2005, pp. 207 – 212.

[16] A. Vavousis, A. Apostolakis, and M. Psarakis, "A fault tolerant approach for FPGA embedded processors based on runtime partial reconfiguration," *Journal of Electronic Testing: Theory and Applications (JETTA)*, pp. 1 – 19, 2013.

[17] A. Sari, M. Psarakis, and D. Gizopoulos, "Combining checkpointing and scrubbing in FPGA-based real-time systems," in *IEEE VLSI Test Symposium (VTS)*, 2013, pp. 1 – 6.

[18] E. Kamanu, P. Reddy, K. Hsu, and M. Lukowaik, "A new architecture for single-event detection and reconfiguration of SRAM-based FPGAs," in *IEEE High Assurance Systems Engineering Symposium (HASE)*, 2007, pp. 291 – 298.

[19] X. Iturbe, M. Azkarate, I. Martinez, J. Perez, and A. Astarloa, "A novel SEU, MBU and SHE handling strategy for Xilinx Virtex-4 FPGAs," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2009, pp. 569 – 573.

[20] M. Palmer, P. Zicari, F. Frustaci, S. Perri, and P. Corsonello, "An efficient and low-cost design methodology to improve SRAM-based FPGA robustness in space and avionics applications," in *International Workshop on Reconfigurable Computing: Architectures, Tools and Applications*, 2009, pp. 74 – 84.

[21] N. Avirneni and A. Somani, "Low overhead soft error mitigation techniques for high-performance and aggressive designs," *IEEE Transactions on Computers*, pp. 488 – 501, 2012.

[22] Z. Qian, Y. Ichinomiya, M. Amagasaki, M. Iida, and T. Sueyoshi, "A novel soft error detection and correction circuit for embedded reconfigurable systems," *IEEE Embedded Systems Letters*, pp. 89 – 92, 2011.

[23] Y. Ichinomiya, M. Amagasaki, M. Iida, M. Kuga, and T. Sueyoshi, "Improving the soft-error tolerability of a soft-core processor on an FPGA using triple modular redundancy and partial reconfiguration," *Journal of Next Generation Information Technology*, pp. 35 – 48, 2011.

[24] S. Rezgui, G. Swift, K. Somervill, J. George, C. Carmichael, and G. Allen, "Complex upset mitigation applied to a re-configurable embedded processor," *IEEE Transactions on Nuclear Science*, pp. 2468 – 2474, 2005.

[25] B. Shashidhara, S. Jadhav, and Y. S. Kim, "Reconfigurable fault tolerant processor on a SRAM based FPGA," in *IEEE International Conference on Electro Information Technology (EIT)*, 2020, pp. 151 – 154.

[26] L. A. C. Benites, F. Benevenuti, . B. De Oliveira, F. L. Kastensmidt, N. Added, V. A. P. Aguiar, N. H. Medina, and M. A. Guazzelli, "Reliability calculation with respect to functional failures induced by radiation in TMR Arm Cortex-M0 soft-core embedded into SRAM-based FPGA," *IEEE Transactions on Nuclear Science*, pp. 1433 – 1440, 2019.

[27] N. Rollins, M. Fuller, and M. Wirthlin, "A comparison of fault-tolerant memories in SRAM-based FPGAs," in *IEEE Aerospace Conference*, 2010, pp. 1 – 12.

[28] N. H. Rollins, "Hardware and software fault-tolerance of softcore processors implemented in SRAM-based FPGAs," Ph.D. dissertation, Brigham Young University, 2012.

[29] Xilinx Corp., "7 series FPGAs memory resources UG473 (v1.14)," Xilinx Documentation, http://www.xilinx.com, 2019.

[30] J. Gomez-Cornejo, A. Zuloaga, I. Villalta, J. Del Ser, U. Kretzschmar, and J. Lazaro, "A novel BRAM content accessing and processing method based on FPGA configuration bitstream," *Microprocessors and Microsystems*, pp. 64 – 76, 2017.

[31] Xilinx Corp., "7 series FPGAs configurable logic block UG474 (v1.8)," Xilinx Documentation, http://www.xilinx.com, 2016.

[32] ——, "Vivado design suite user guide. partial reconfiguration UG909 (v2019.2)," Xilinx Documentation, http://www.xilinx.com, 2020.

[33] A. Morales-Villanueva, R. Kumar, and A. Gordon-Ross, "Configuration prefetching and reuse for preemptive hardware multitasking on partially reconfigurable FPGAs," in *Design, Automation & Test in Europe (DATE)*. IEEE, 2016, pp. 1505 – 1508.

[34] J. Gomez-Cornejo, A. Zuloaga, U. Bidarte, J. Jimenez, and U. Kretzschmar, "Interface tasks oriented 8-bit soft-core processor," in *Annual FPGAworld Conference*, 2012, pp. 4:1 – 4:5.