Criticality-aware Scrubbing Mechanism for SRAM-based FPGAs

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

***Abstract*—Scrubbing has been considered as an effective mechanism to provide fault-tolerance in Static-RAM (SRAM)-based Field Programmable Gate Arrays (FPGAs). However, the current scrubbing techniques execute without considering the criticality and timing of the user tasks implemented in the FPGA. They often do not execute the scrubbing process in the right instant, which minimizes the probability of each task being executed without transient faults. Moreover, these current solutions are not adapted to the tasks' fault-tolerance requirements, since they may not properly protect the most critical tasks in the system. However, if they do it, they waste resources with the less critical tasks. In this paper, a new scrubbing mechanism is proposed. This new approach adapts the scrubbing mechanism to the tasks' execution, by a proper scheduling and according to their criticality. A proposed heuristic finds a feasible scrubbing schedule for each hardware task. Firstly, the minimum scrubbing periods are computed according to the criticality of each implemented hardware task. Secondly, a proper scrubbing schedule following the EDL (Earliest Deadline as Late as possible) algorithm is found, maximizing the reliability of the system. The experimental results show up to 79% improvements on the system reliability, achieved without wasting scrubbing resources.**

I. Introduction

Nano-satellites have been gaining significant importance in space missions, since traditional satellites are expensive to launch. They have a lighter weight, usually under ten kilograms and a smaller size. This enables the use of smaller and more efficient launch vehicles, thereby reducing the launch costs. Nano-satellites have empowered smaller countries and research institutes to explore and obtain data from space, leading to a reduction in the development and production costs. For instance, the use of Commercial off-the-shelf (COTS) devices and associated tools allow a faster development and a lower price due to the economy of scale.

Following this trend, COTS Static-RAM (SRAM)-based to Field Programmable Gate Arrays (FPGAs) have been significantly used in space environments, since they present a higher operational capacity and performance, as well as reconfiguring and reprogramming capabilities. In particular, reconfiguring and reprogramming properties are obviously very useful after the devices have been launched into space. However, in space, FPGAs are commonly affected by charged particles that strike the silicon substrate. These events called Single Event Upsets (SEUs) can inadvertently change the device outputs and corrupt the function results. Hardened FPGAs are one way to mitigate this problem, but they are very expensive. Due to the high price, several fault-tolerance mechanisms have been developed in order to increase the reliability of SRAM-based FPGAs. Scrubbing is one of these mechanisms that take advantage of the FPGA reconfiguration capability. Scrubbing mechanisms periodically verify the FPGA configuration memory for faults. When a fault is detected, they correct it using the original data, stored in an external memory.

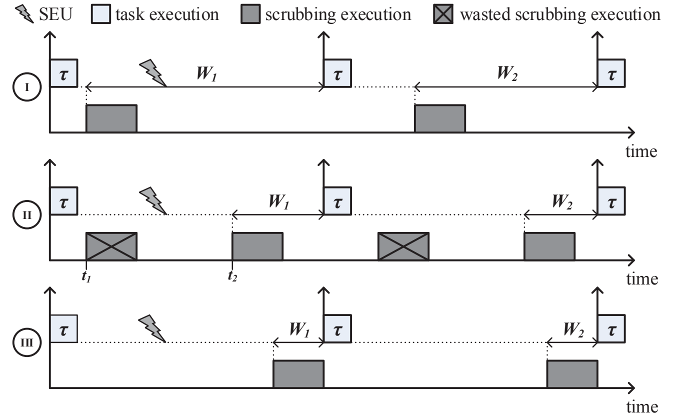

Fig. 1. Motivation example.

A common drawback among the current scrubbing solutions is the independence of the scrubbing execution and the hardware tasks implemented on the FPGA. The entire FPGA configuration memory is scrubbed sequentially at a constant rate without any relation to the importance and timing of tasks executed on it. In order to better understand this mechanism, let's consider the simple example described in Figure 1. A task τ with a periodic behaviour is implemented on an FPGA device, using a certain amount of hardware resources. These resources are scrubbed periodically as described in Figure 1-I. Since there is no relation between task τ and its scrubbing executions, the gap between the several task τ jobs and the last corresponding scrubbing can be significantly large (w_1 and w_2 intervals), increasing the probability of the task τ encountering a transient fault, which will affect its functional outputs. Another drawback is when the scrubbing period is smaller than the task τ period. Considering the example described in Figure 1-II, the first scrubbing execution (at instant t_1) is totally irrelevant for the reliability of the task τ, since any fault between the instants t_1 and t_2 can be handled by the second scrubbing execution (one that starts at instant $t2$). In this case, scrubbing resources are wasted. The example in Figure 1-III shows the ideal instant to scrub the FPGA configuration memory related to task τ that minimizes the probability of an SEU affecting its jobs' execution.

Nowadays, FPGA systems can accommodate a large number of heterogeneous tasks with different timing and fault-tolerance requirements, which consequently organizes them in different levels of criticality or importance for the system [1]. One fault in a very critical task may have a significant negative impact on the system as compared to a fault in a less critical task. Using a decoupled scrubbing process from the tasks' execution, the current scrubbing solutions are not efficient for these systems. They may not properly protect the most critical

tasks if the scrubbing period is higher than the tasks' period. On the other hand, if they scrub the critical tasks with the shortest possible period, they are wasting scrubbing resources (time) with the less critical tasks. Wasted scrubbing resources can also mean wasted power, which is limited and critical in space equipments.

Contributions: This paper proposes an efficient scrubbing mechanism, which takes into account the criticality and timing of the hardware task execution in order to increase the reliability of the system. A heuristic that finds a feasible scrubbing scheduling for each hardware task is introduced. Firstly, the minimum scrubbing periods are computed according to the criticality of each implemented hardware task. Secondly, a proper scrubbing schedule following the EDL (Earliest Deadline as Late as possible) algorithm is found, maximizing the reliability of the system. With this approach, experimental results show up to 79% of improvement on the system reliability without wasting scrubbing resources.

To the best of the authors' knowledge, this is the first work that proposes a schedule for the scrubbing of tasks, taking into account the hardware tasks' importance and timing (mapped to the fault-tolerance requirements) in order to increase the reliability of the system.

The rest of the paper is organised as follows. Section II presents the FPGA background and related works concerning the scrubbing mechanisms. Section III describes the system model. Section IV details the problem and deduces its complexity. Section V proposes a heuristic to solve it. Section VI presents a case study. Section VII introduces the experiments performed and discusses the respective results. Finally, Section VIII presents the conclusions and the future work.

II. Background and Related Work

A brief background on SRAM-based FPGAs is first presented before discussing the related works on scrubbing.

A. FPGA Background

One of the most important features provided by the current FPGAs is their ability to be reconfigured at runtime, modifying both the structure and functionality of the implemented circuit. This is achieved through the use of the Internal Configuration Access Port (ICAP), which reads back the current circuit and can write back a different configuration at runtime. An FPGA can accommodate multiple hardware/software tasks each placed in its own partition of the FPGA (region), commonly called partially reconfigurable regions (PRRs). Each partition is responsible for implementing a specific hardware functionality, which can be dynamically modified by loading the respective bit file, while the other blocks continue their normal operation. Different circuits with different functionalities (different bit files) can be loaded onto the FPGA and thereafter be used for the reconfiguration when required. Each PRR is composed of several configuration frames and each configuration frame contains the bitstream of several reconfigurable digital blocks, such as Look-Up Tables (LUTs), Block-RAMS (BRAMs), Flip-Flops (FFs), switch matrices, etc. If a digital block in a particular configuration frame needs to be reconfigured, the entire frame has to be rewritten. In this sense, a configuration frame is the lowest reconfigurable granularity that can be found in an FPGA. For example, the Xilinx Virtex-6 FPGA (XC6VLX240T-1f1156) contains $28,464$ frames in total and each frame contains $2,592$ bits (81 words).

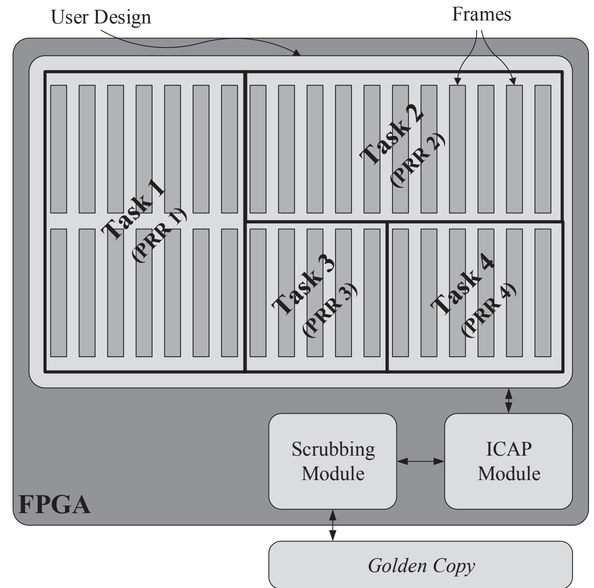

Fig. 2. FPGA – background.

B. Scrubbing Related Work

Scrubbing is a mechanism to repair faults on the FPGA that takes advantage of the frame reconfiguration. Several fault-tolerance solutions have been developed around this mechanism with the simplest approach being blind scrubbing [2] [3]. This solution does not detect the existence of faults on the FPGA, but it periodically rewrites the configuration frames (bitstream file) onto the FPGA instead, overwriting possible faulty bits caused by SEUs. An external memory with continuous access is required to store the original configuration frames, frequently called *golden copy* (Figure 2). Readback scrubbing is another solution, which enables fault detection, reading frame-by-frame the configuration data from the FPGA and then performing a bit-for-bit comparison to the original frames stored in the external memory (*golden copy*). Another alternative combines readback scrubbing with Error Correction Codes (ECCs) [4]–[6]. This approach enables fault detection by reading the configuration data frame-by-frame and computing their error correction codes (ECCs) and comparing them to the original ones previously computed and stored externally for each frame. Nazar *et al.* [7] also introduce an improvement in the scrubbing mechanism, assuming a non-uniform distribution of the critical bits in a given partition. In this sense, some regions are more likely to have a fault affecting the system. With this information, the authors propose a mechanism that statistically finds the optimal frame to start the scrubbing, which reduces the mean time to repair a certain fault.

All these scrubbing mechanisms are independent of the hardware tasks implemented in the FPGA. The entire FPGA is scrubbed sequentially with a constant rate and considering an initial starting point. Therefore, the time interval between the scrubbing and the task execution can be large, increasing the probability of the task being affected by a fault. Moreover, they do not provide differentiated scrubbing rates according to the criticality of the tasks in the system. The most critical tasks may not obtain the sufficient scrubbing resources in order to reach a certain reliability. On the other hand, if they are scrubbed with the shortest possible period, scrubbing resources are wasted with the less critical tasks.

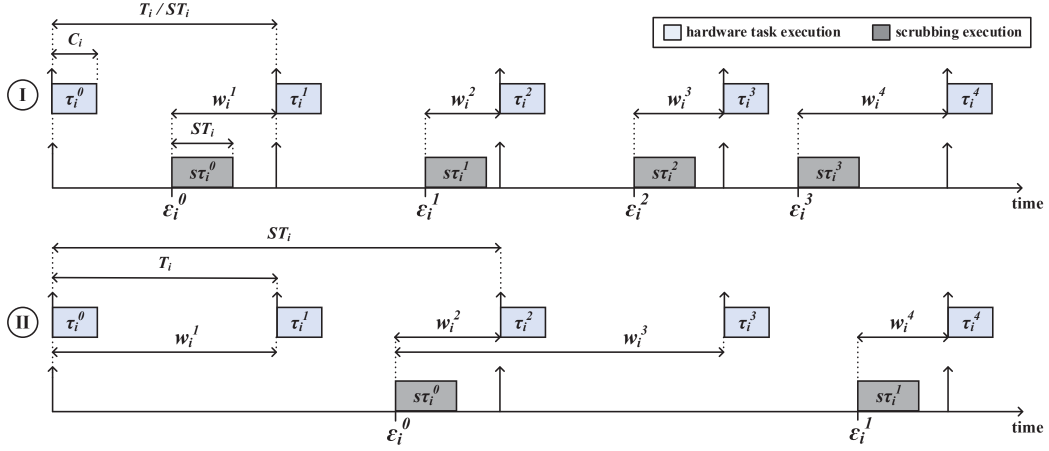

Fig. 3. Hardware task execution and proposed scrubbing execution. I) Scrubbing executed every task execution instance. II) Scrubbing executed every two task execution instances.

III. SYSTEM MODEL

A. Task Model

An FPGA device can accommodate several functionalities (tasks) divided by a set of partitions SP, as described in Figure 2. Each partition, also called partial reconfigurable region, $PRR_i \in SP$ is responsible for implementing a specific hardware functionality. Many of these functionalities have a periodic behaviour and, in this sense, they are generally modelled as a set Γ of periodic tasks. Each task $\tau_i \in \Gamma$ implemented on the PRR_i is characterized by four parameters $\tau_i = (C_i, T_i, \eta_i, \zeta_i)$: C_i defines the worst case execution time in hardware; T_i represents the execution period; η_i defines the number of FPGA configuration frames used to implement the task; and finally, ζ_i represents the criticality of the task in the system (higher values of criticality mean higher criticality). Their execution described in Figure 3-I can be interpreted as following: τ_i is released in an infinite sequence of jobs at the instants kT_i, with $k = \{0, 1, 2, 3, ...\}$. The job instance τ_i^k has to execute C_i time units during the interval $[kT_i, (k+1)T_i)$, assuming the deadline is equal to the period T_i. Once, these tasks execute in their own dedicated hardware, they do not suffer any interference or blocking from other tasks in the system. Therefore, we can assume that each job τ_i^k is executed without jitter at the beginning of the period, i.e, at the instant kT_i.

The proposed scrubbing mechanism does not scrub the entire FPGA or the used configuration frames sequentially with a constant rate, but it scrubs the FPGA configuration frames associated to each task τ_i independently from the others, enabling adaptive scrubbing. The FPGA configuration frames associated to task τ_i are scrubbed taking into account task τ_i's execution as well as its criticality. In this sense, there is a scrubbing task set $s\Gamma$ associated to the task set Γ. Each scrubbing task $s\tau_i \in s\Gamma$ represents the scrubbing process of task τ_i, as described in Figure 3-I. Moreover, each scrubbing task $s\tau_i$ can also be modelled as a periodic task characterized by three parameters $s\tau_i = (SC_i, ST_i, \zeta_i)$: SC_i defines the time to scrub the η_i FPGA frames used to implement the task τ_i; ST_i represents the scrubbing period, which is a multiple of the corresponding task period T_i; and finally ζ_i is the criticality of the task τ_i. The execution of $s\tau_i$ can be interpreted as following: $s\tau_i$ is released in an infinite sequence of jobs at the instants pST_i, with $p = \{1, 2, 3, ...\}$. Note that the release of the first scrubbing job is synchronized with the corresponding first task job execution. Considering the deadline of $s\tau_i$ to be equal to its period ST_i, the job instance $s\tau_i^p$ has to execute SC_i time units without preemption during the interval $[pST_i, (p+1)ST_i)$, as described in Figure 3-I. Non-preemption is considered in order to reduce the scheduling overhead and to minimize the size of the stored scrubbing schedule table.

B. Error Model

As referred above, the FPGA device composed by Θ configuration frames can be affected by SEUs, which follow a Poisson distribution with a rate of λ failures per unit of time [8]. Therefore, the probability of no failures in an unprotected FPGA-based design in an interval t is given by the following equation [9] [10].

$$P_{ne} = e^{-\frac{\lambda \Theta}{\Theta} t} \Leftrightarrow P_{ne} = e^{-\lambda t} \tag{1}$$

For a given task τ_i, the ideal scrubbing instant, i.e., the instant that reduces the probability of the kth job to be executed without suffering any fault, is the interval immediately before its executions. This mechanism reduces the probability of a task execution being affected by any fault. Therefore, the probability of kth job execution of the task τ_i being executed without any fault, considering the last corresponding $s\tau_i$ job execution, is given by:

$$P_{ne}[\tau_i^k] = e^{-\frac{\lambda \eta_i}{\Theta} w_i^k}, \tag{2}$$

where w_i^k is the time interval between the beginning of the last scrubbing process and the beginning of the kth job execution, as described in Figure 3-I. Note that the SEU rate that affects a task τ_i is proportional to the hardware resources (configuration frames) used by it. Moreover, faults during each task execution instance are not considered.

Definition 1. (RELIABILITY OF A TASK) *Reliability of a task is a metric that defines the probability of a task τ_i being executed in the interval $[0, t]$ without faults [11].*

Therefore, the reliability of the task τ_i can be expressed in the following equation as the probability of all instances of τ_i in the interval $[0, t]$ being executed without faults.

$$R_i(t) = P_{ne}[\tau_i^0 \wedge \tau_i^1 \wedge ... \wedge \tau_i^k], \quad kT_i \leq t \qquad (3)$$

Definition 2. (SYSTEM RELIABILITY BASED ON CRITICALITY) *System reliability based on criticality is a metric that defines the system reliability during the interval $[0, t]$ taking into account the reliability as well as the criticality of each task executing in the system.*

Therefore, the system reliability based on criticality can be expressed as follows,

$$R(t) = \sum_{i=0}^{|\Gamma|-1} R_i(t) \times norm_1(\zeta_i) \qquad (4)$$

where $|\Gamma|$ is the number of tasks that are executing in the system and $norm_1(\zeta_i)$ is the normalized task criticality.

IV. Problem Definition

The ideal case, which maximizes the reliability of each task τ_i during an interval $[0, t]$ is to scrub the corresponding FPGA frames in all job instances of τ_i and just before their execution as described in Figure 3–I. However, this ideal case may not be possible to achieve, since the ICAP resources (time to read and write the user design frames) are limited. In that case the scrubbing periods (ST_i) may have to be larger than the task periods (T_i), as described in Figure 3–II.

Taking into account these concerns, the problem can be formulated as follows. Given all hardware tasks τ_i implemented in FPGA using a certain number of configuration frames, which require SC_i time units to be scrubbed, we want to determine the exact scrubbing schedule (ε_i^p and ST_i – refer to Figure 3), in way that it enables the fault-tolerance in all tasks in the system and reduces the probability of each task job execution τ_i^k being affected by a fault. The most critical hardware tasks in the system must be the most reliable tasks, i.e., they must have the lowest probability of being affected by a fault, while the less critical tasks may be less reliable. In short, the global objective is to maximize the system reliability based on the task criticality. Therefore, taking equations 3 and 4 into account, the objective function can be expressed as follows,

$$max\ R(t) = max \sum_{i=0}^{|\Gamma|-1} \sum_{k=0}^{\lfloor t/T_i \rfloor} P_{ne}[\tau_i^k] \times norm_1(\zeta_i) \qquad (5)$$

where $|\Gamma|$ is the number of tasks in the system and $\lfloor t/T_i \rfloor$ gives the number of hardware task τ_i instances in a pre-defined time interval $[0, t]$.

A. Complexity

This scrubbing problem can be modelled as a well known non-preemptive scheduling problem. However, these problems are NP-Hard in the strong sense as Jeffay *et al.* [12] have showed. In the next section, a heuristic is proposed in order to find a feasible solution in a suitable time interval.

V. Proposed Heuristic

If the scrubbing tasks are scheduled as close as possible to their deadlines (Figure 3), the probability of the corresponding hardware tasks being affected by a fault is reduced. Therefore, instead of finding the exact scheduling for the scrubbing tasks, we propose a heuristic that finds the minimum scrubbing periods which makes the scrubbing task set schedulable by the utilization-based schedulability test. Then, the earliest deadline as late as possible (EDL) algorithm [13] is used to schedule the scrubbing tasks, since it schedules the tasks as late as possible near the deadlines. However, EDL algorithm is not optimal for non-preemptive task sets. Although the task set verifies the utilization-based schedulability condition given by equation $\sum_{i=0}^{s\Gamma-1} \frac{SC_i}{ST_i} \leq 1$, when the schedule is performed, there may be one or more tasks which do not meet their deadlines. Therefore, in order to find the minimum periods which make the scrubbing process schedulable, an iterative approach is used. If the scrubbing task set is not schedulable, the process is repeated after reducing the utilization upper bound, until the scrubbing task set is scheduled.

Algorithm 1 describes the proposed heuristic. Three main steps can be highlighted. The first step implemented by the function $findSPeriods$ (line 4) finds the minimum periods, which makes the scrubbing task set $s\Gamma$ schedulable according to the utilization-based schedulability test. This step is performed using an Integer Linear Programming (ILP) formulation described in the following subsection. The second step, implemented by the function $computeLCM$ (line 5), computes the least common multiple (LCM) of the obtained periods in the previous step. The schedule produced in the next step has a cyclic property, i.e, for every LCM interval, the schedule is repeated [14]. Therefore, the feasibility of the schedule only has to be assured for the LCM interval. The third step implemented by the function $edlSchedule$ (line 6) verifies the schedulability of the system following the EDL algorithm [13] and returns the corresponding schedule. Basically, the schedule of the scrubbing tasks is built and at the same time the schedulability is verified. Note that the EDL scheduling algorithm is not optimal for non-preemptive task sets. Therefore, if the scrubbing task set is not schedulable, new scrubbing periods have to be computed. The utilization upper bound is reduced (line 7) by Δ, given as an algorithm input variable. This process is repeated until the scrubbing task set is scheduled. When a schedulable scrubbing task set is found, the corresponding schedule is returned (line 9).

Algorithm 1 Proposed Heuristic

Require: Γ, $s\Gamma$, Δ;
Ensure: schedule;
1: uBound = 1;
2: is_schedulable = 0;
3: **while** is_schedulable == 0 **do**
4: findSPeriods(Γ, $s\Gamma$, uBound); *//using ILP*
5: lcm = computeLCM($s\Gamma$);
6: (is_schedulable, schedule) = edlSchedule($s\Gamma$, lcm);
7: uBound = uBound – Δ;
8: **end while**
9: **return** schedule;

A. ILP formulation

The scrubbing task periods returned by the function $findSPeriods$ in Algorithm 1 are computed by an ILP formulation. As described, the input variables are the hardware task

TABLE I. TASK SET (Γ) PARAMETERS

τ_i	Task Name	$C_i(ms)$	$T_i(ms)$	η_i	ζ_i
τ_0	$Control_Law$	1.2	50	250	8
τ_1	$Process_IRES_data$	0.41	100	150	7
τ_2	$Calibrate_Gyro$	0.39	100	100	6
τ_3	$Present_Encryptor$	1.0	10	1200	2
τ_4	$MPEG4_Encoder$	1.0	10	800	1

set (Γ), scrubbing task set ($s\Gamma$) and the system utilization upper bound ($uBound$). Given these inputs, the ILP formulation finds the scrubbing periods ST_i, multiples of the corresponding task periods T_i,

- ϕ_i := ratio between the scrubbing and the task execution period ($ST_i = \phi_i \times T_i$).

The scrubbing periods (ST_i) are computed according to the objective function defined in the following equation, which minimizes the $s\tau_i$ scrubbing periods (ST_i) according to the criticality (ζ_i) of the corresponding task τ_i. Therefore, the less critical tasks have higher scrubbing period (ST_i). On the other hand, the more critical tasks have smaller scrubbing periods.

- $$min \sum_{i=0}^{|\Gamma|-1} ST_i \times \zeta_i$$

The objective function is subjected to the presented utilization-based schedulabilty condition:

- $$\sum_{i=0}^{|\Gamma|-1} \frac{SC_i}{ST_i} <= uBound$$

VI. CASE STUDY

This case study highlights two subsystems of a nano-satellite, namely the navigation control and the payload subsystem. The navigation control subsystem is responsible for monitoring and controlling the orientation of the nano-satellite, while the payload subsystem is responsible for implementing the functionality of the nano-satellite.

The considered nano-satellite collects high resolution video images from the earth. These video images are encrypted before being sent to a ground station. Five processing tasks were implemented in an SRAM-based FPGA, whose parameters are defined in Table I. The first three tasks, based on [15], are related to the nano-satellite navigation subsystem and hence have a higher criticality. The other two tasks are related to the payload subsystem and have a lower criticality. The $Process_IRES_data$ processes the information data provided by the infra-red earth sensor, which is required to stabilize the satellite in the right orbit. The $Control_Law$ task implements the space control laws that are responsible for the satellite orientation and its results are provided to the actuators. $Calibrate_Gyro$ computes the necessary information in order to properly calibrate the gyroscope. $MPEG4_Encoder$ [16] encodes the video frames captured by the nano-satellite camera to be transmitted to the earth, while $Present_Encryptor$ [17] encrypts the encoded video using the Present algorithm.

The five tasks were implemented in an SRAM-based FPGA with $30,000$ configuration frames[1]. It was assumed that each frame of the FPGA device requires $1\mu s$ to be scrubbed. The FPGA device was targeted to be placed in the space environment, subjected to SEUs with a rate $\lambda = \frac{1}{1Hour}$ [8]. Moreover, in order to reduce the energy consumption, the maximum percentage of ICAP utilization was limited to 20%.

[1]Comparable to Xilinx Virtex-6 FPGA board.

TABLE II. SCRUBBING TASK SET ($s\Gamma$) PARAMETERS

$s\tau_i$	$SC_i(ms)$	$ST_i(ms)$	ζ_i
$s\tau_0$	0.25	50	8
$s\tau_1$	0.15	100	7
$s\tau_2$	0.10	100	6
$s\tau_3$	1.20	10	2
$s\tau_4$	0.8	20	1

TABLE III. CASE STUDY RESULTS

Method	System Reliability	Wasted ICAP Time (s)
Proposed	0.99	0
Selective	0.90	1170
Blind	0.58	6600

According to the proposed scrubbing solution, a scrubbing task ($s\tau_i$) must be defined for each hardware task (τ_i). Therefore, the minimum scrubbing periods, which make the scrubbing task set schedulable by the EDL algorithm, using 20% of the ICAP module, have to be obtained. Table II presents the scrubbing tasks' parameters given by the proposed heuristic (in particular the scrubbing periods – ST_i). Note that in order to have a scrubbing task set that uses only 20% of the ICAP module capacity, task τ_4, the less critical one, will be scrubbed one time in two periods ($ST_4 = 20ms$). The remaining tasks will be scrubbed every period.

We consider two different types of scrubbing for the case study – blind scrubbing and selective scrubbing. Blind scrubbing scrubs the entire device without considering the tasks that have been implemented on it. However, selective scrubbing only scrubs those frames of the device that are utilized by the user design. Selective scrubbing has a better reliability than blind scrubbing since only those frames that are essential to the user design are scrubbed.

Figure 4 describes the tasks' execution as well as the scrubbing execution according to the proposed and the current scrubbing mechanisms. Note that with 20% of ICAP capacity available, each task is scrubbed every $12.5ms$ using the selective approach and every $150ms$ using the blind one. The proposed scrubbing mechanism executes the scrubbing tasks following the EDL algorithm, i.e, as late as possible, nearest to the deadline (ST_i). Therefore, the scrubbing tasks execute just before the hardware task execution, minimizing the probability of the task jobs being affected by an SEU fault. On the other hand, the existing scrubbing approaches execute without any relation to the task execution, decreasing the system reliability. Also Figure 4 shows the wasted resources for both selective and blind scrubbing.

Table III presents for this particular task set the system reliability computed through the equation 4. The system reliability was measured for a duration of 10 hours. Using the proposed approach, the system is 9% and 41% more reliable when compared to selective scrubbing and blind scrubbing, respectively. Taking into account the ICAP wasted resources, during 10 hours selective scrubbing wasted $1,170$ seconds of ICAP utilization and selective scrubbing wasted $6,600$ seconds.

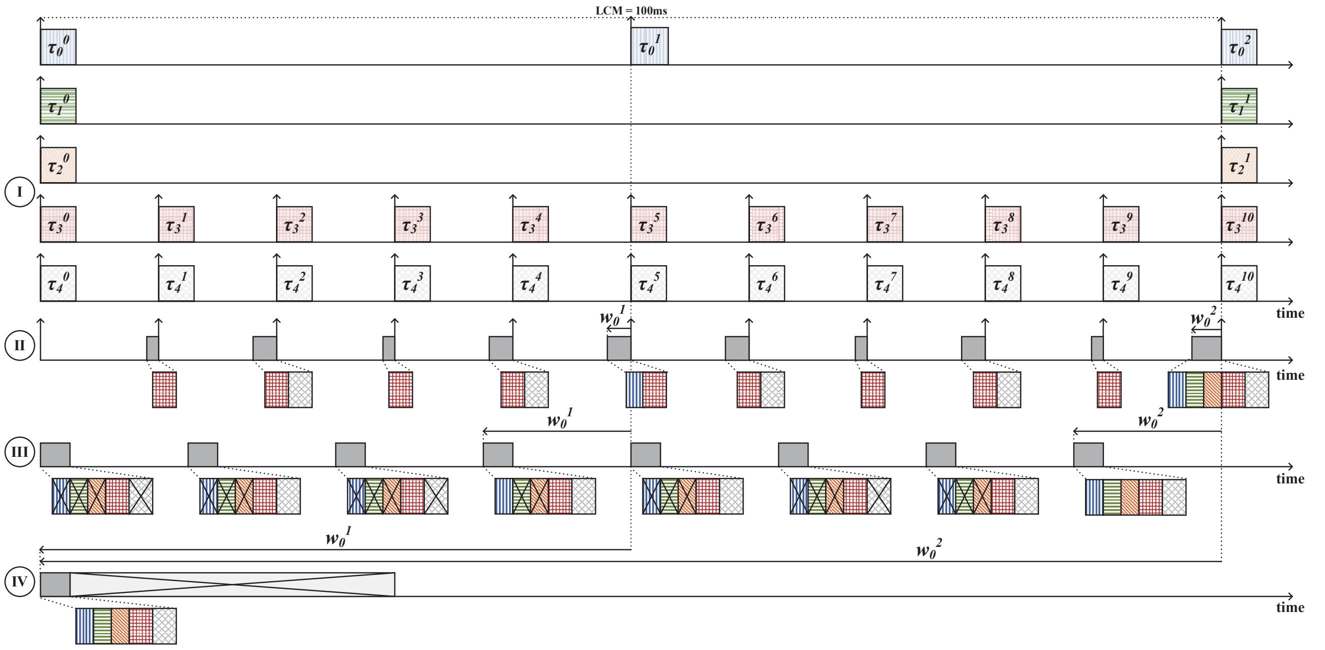

Fig. 4. I) Case study hardware task execution. II) Proposed scrubbing execution. III) Selective scrubbing execution IV) Blind scrubbing execution. Notes: 1) graphs are not in scale; 2) wasted scrubbing execution marked with a cross; 3) time intervals between the scrubbing execution and the task τ_0 execution are shown for the different scrubbing mechanism.

VII. EXPERIMENTAL RESULTS

Experiments were conducted in order to better evaluate the new scrubbing mechanism. The experiments were also based on SRAM-based FPGA with 30,000 configuration frames[2]. It was assumed that each frame requires $1\mu s$ to be scrubbed by the ICAP module. Moreover, for these experiments the FPGA device was also targeted to be placed in the space environment, subjected to SEUs with a rate $\lambda = \frac{1}{1Hour}$ [8].

The experiments used task sets with different sizes, ranging from 1 to 20 tasks, which generated different loads of used resources (configuration frames) in the FPGA. For each task set size, $1,000$ synthetic hardware task sets were generated. The number of configuration frames used to implement each task τ_i assumes only values that are multiples of 100, synthetically generated and uniformly distributed from $1,000$ and $2,000$, corresponding to a scrubbing execution time (SC_i) between $1ms$ and $2ms$, respectively. Moreover, the period (T_i) of each task τ_i assumes only values that are multiples of $5ms$, synthetically generated and uniformly distributed between $10ms$ and $50ms$.

The proposed scrubbing mechanism was compared to the blind scrubbing and the selective scrubbing mechanism, described in the previous section. Moreover, the proposed mechanism considered three different methods to assign the task criticality. The first method ($i^0 = 1$) assigns the same criticality to all the tasks τ_i. The second method (i^1) assigns to the task τ_i the criticality i, while the third method (i^2) assigns the criticality i^2.

A. System reliability over the number of tasks implemented

The top graph in Figure 5 shows, over the task set size, the amount of resources (configuration frames) used. Moreover, it shows the ICAP utilization generated by the different scrubbing mechanisms. Note that the percentage of configuration frames used increases linearly with the size of the task set, reaching the full capacity of the FPGA at 20 tasks. Moreover, the scrubbing mechanisms were configured to use 100% of ICAP module capacity. The proposed mechanism increases the ICAP utilization with the size of the task set, reaching 100% at 17 tasks. Blind and selective scrubbing scrub at the maximum allowed ICAP capacity (100%), regardless of the number of tasks implemented.

The middle graph in Figure 5 compares the system reliability among the proposed, the blind and the selective scrubbing mechanisms over the task set size and a duration of 10 hours. The proposed mechanism is evaluated considering the different methods to assign the task criticality, as described above. We observe that for task sets with only one task, the proposed mechanism performs the same way as selective scrubbing. However, selective scrubbing wastes a huge amount of ICAP resources, since it is constantly scrubbing the implemented task, as presented latter. In comparison to blind scrubbing the proposed mechanism improves the system reliability around 33% without wasting scrubbing resources. Between 1 and 20 tasks the reliability of the system decreases linearly. This is due to the increased interference among the scrubbing tasks when they are scheduled by the EDL algorithm. With 20 tasks, the FPGA is completely used, therefore the maximum interference is reached. In this case, the proposed scrubbing mechanism improves the system reliability 5%, 10%, and 14% when compared to selective scrubbing; and 10%, 15% and 19%, when compared to blind scrubbing and using the different methods to assign the criticality.

The bottom graph in Figure 5 shows the wasted ICAP resources by the different scrubbing mechanisms, measured also during 10 hours. The proposed scrubbing mechanism has not wasted ICAP resources, since the scrubbing is only

[2]Comparable to Xilinx Virtex-6 FPGA board.

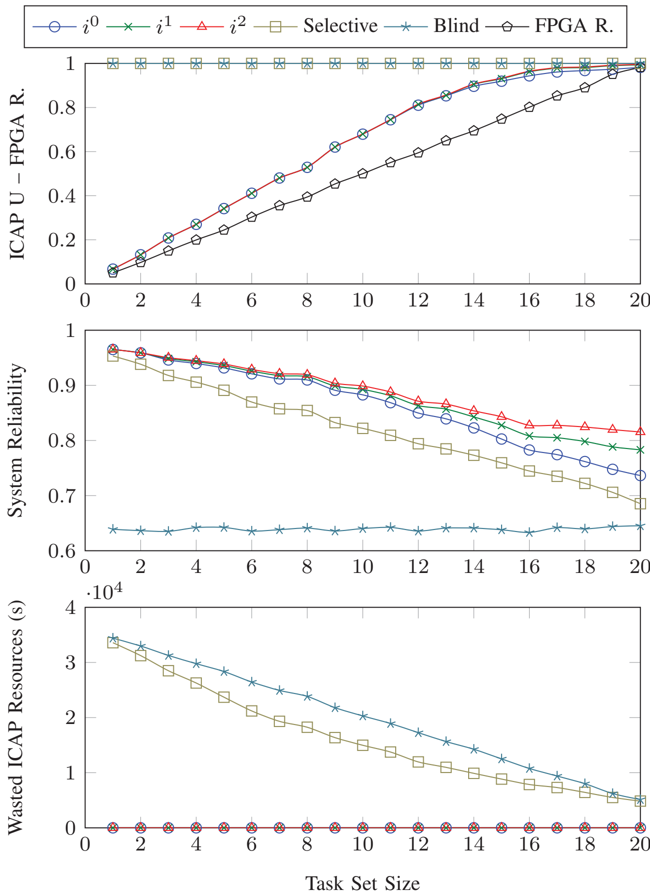

Fig. 5. System reliability – Maximum ICAP utilization = 100%.

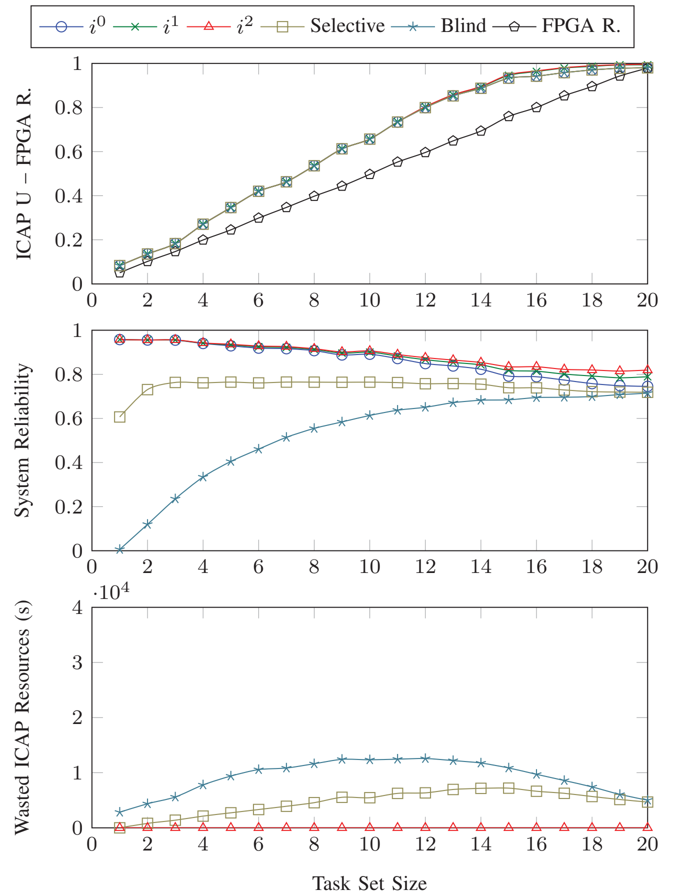

Fig. 6. System reliability – Selective and Blind scrubbing mechanisms using the same ICAP utilization as the one used by the proposed mechanism (i^0).

performed when the hardware tasks execute. On the contrary, selective and blind scrubbing present a huge amount of wasted scrubbing resources, since they are not adapted to the hardware task execution. For instance, for 1 task, selective scrubbing wastes $34,300$ seconds of ICAP utilization and blind scrubbing wastes $34,750$ seconds. In embedded systems, in particular in space equipments, this wasted resources can have a great impact on the power consumption.

For a fairer comparison, the selective and blind approaches instead of using the maximum allowed ICAP capacity (100%) only use the ICAP utilization generated by the proposed scrubbing mechanism, in particular the one generated using the method i^0. The graphs in Figure 6 present the results. For task sets with one task there is an improvement on the system reliability of 36% and 96% when comparing the proposed solution to the selective and blind scrubbing, respectively. For task sets with 20 tasks, as expected, the results are the same as the last experiment, since selective and blind approaches are also allowed to use the maximum ICAP capacity (100%).

B. System reliability over the ICAP utilization

The graph in Figure 7 shows the system reliability (equation 4) for a task set size with 5 tasks implemented over maximum ICAP utilization allowed, measured for a duration of 10 hours. The ICAP utilization allowed ranges from 20% and 100%. As expected the system reliability decreases when the ICAP capacity used by the scrubbing mechanism decreases.

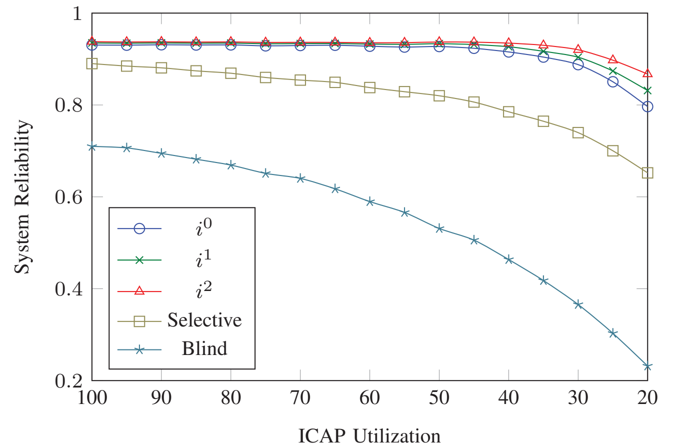

Fig. 7. System reliability based on the criticality over the ICAP utilization.

If the scrubbing mechanism is configured to use up to 100% of the ICAP capacity, the proposed solution performs 5% and 28% better when compared to selective and blind scrubbing respectively. On the other hand, if the scrubbing mechanisms only uses 20% of the ICAP capacity, the proposed approach performs between 14% and 24% better when compared to selective scrubbing. In comparison to blind scrubbing, it performs between 56% and 64% better.

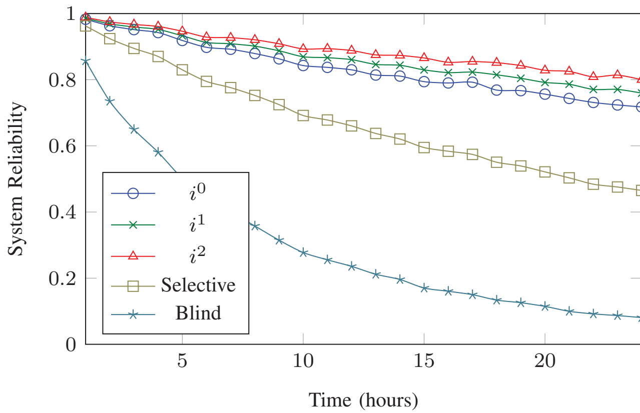

Fig. 8. System reliability based on the criticality over the timing interval.

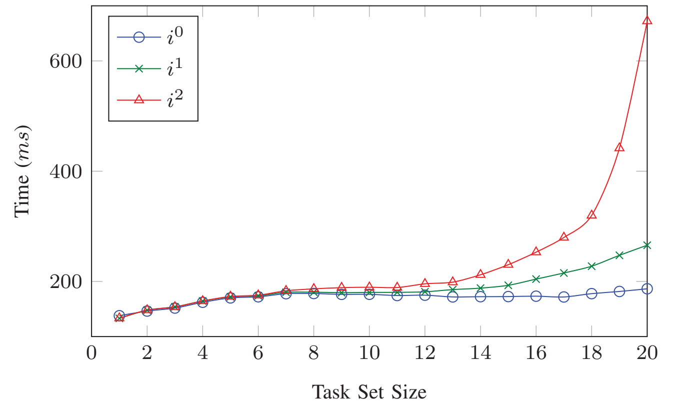

Fig. 9. Heuristic time latency (experiment conducted on a quad-core Intel i7-3770 – 3.40GHz).

C. System reliability over the timing interval

The graph in Figure 8 shows the system reliability (equation 4) for a task set size with 5 tasks implemented over the size of the time interval used to measure the reliability. The ICAP capacity provided to the scrubbing mechanisms is 25%. As expected the system reliability decreases linearly with the time interval size used to measure the system reliability. For a 24 hour window size the proposed mechanism performs between 25% and 34% better than the selective mechanism. It also performs between 70% and 79% better than the blind scrubbing, considering the different methods to assign the criticality (i^0, i^1, i^2).

D. Heuristic performance

Considering the first experiment with an ICAP utilization equal to 100%, the graph in Figure 9 shows an average for each task set size, the time taken by the proposed heuristic to produce a feasible schedule, using the three different methods to assign the task criticality (i^0, i^1, i^2). Note that, this time is dependent on the generated task sets, in particular on the task periods (T_i), since they can set scrubbing periods multiples of (T_i), having impact on the LCM. For higher ICAP utilizations i^2 takes more time to find a feasible schedule, since it has to scrub more frequently the most critical tasks, increasing the scrubbing periods of the less critical ones. In this case the LCM may increase as well as the time to verify the schedulability, as a consequence. Thus, as shown in the graph, a feasible schedule is obtained in a suitable time. In the same experiment, in $20,000$ task sets generated, only 100 task sets required more than 1 heuristic iteration to find a feasible scrubbing schedule.

VIII. Conclusions and Future Work

In this paper, a new scrubbing mechanism is proposed in order to improve the system reliability. This new mechanism takes into account each hardware task execution as well as its criticality in the system to find the proper scrubbing period which maximizes the overall system reliability when all the scrubbing processes of all tasks are scheduled by the EDL algorithm. Experiments conducted show effective improvements up to 79% on the system reliability without wasting scrubbing resources, when compared to the current scrubbing approaches. In the near future, we will study the impact on the system reliability and on the scheduling overhead of a preemptive approach (scheduling on the frame level). Moreover, we will study the impact of the wasted ICAP resources on the FPGA power consumption.

References

[1] S. Baruah, H. Li, and L. Stougie, "Towards the Design of Certifiable Mixed-criticality Systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'10)*, 2010.

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

[3] J. Heiner, B. Sellers, M. Wirthlin, and J. Kalb, "FPGA partial reconfiguration via configuration scrubbing," in *IEEE International Conference on Field Programmable Logic and Applications (FPL'09)*, 2009.

[4] C. Argyrides, D. Pradhan, and T. Kocak, "Matrix Codes for Reliable and Cost Efficient Memory Chips," *IEEE Transactions on Very Large Scale Integration Systems (VLSI'11)*, 2011.

[5] S. P. Park, D. Lee, and K. Roy, "Soft-Error-Resilient FPGAs Using Built-In 2-D Hamming Product Code," *IEEE Transactions on Very Large Scale Integration Systems (VLSI'12)*, 2012.

[6] 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 (ISIE'10)*, 2010.

[7] G. Nazar, L. Santos, and L. Carro, "Accelerated FPGA Repair Through Shifted Scrubbing," in *IEEE International Conference on Field Programmable Logic and Applications (FPL'13)*, 2013.

[8] C. C. Brendan Bridgford and C. W. Tseng, *Single-Event Upset Mitigation Selection Guide*, ser. Xilinx Corporation, 2008.

[9] A. Das, A. Kumar, and B. Veeravalli, "Aging-aware Hardware-software Task Partitioning for Reliable Reconfigurable Multiprocessor Systems," in *IEEE International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'13)*, 2013.

[10] P. Axer, M. Sebastian, and R. Ernst, "Reliability Analysis for MPSoCs with Mixed-critical, Hard Real-time Constraints," in *IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'11)*, 2011.

[11] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, 1st ed. Norwell, MA, USA: Kluwer Academic Publishers, 1997.

[12] K. Jeffay, D. Stanat, and C. Martel, "On non-preemptive scheduling of period and sporadic tasks," in *IEEE International Real-Time Systems Symposium (RTSS'91)*, 1991.

[13] H. Chetto and M. Chetto, "Some Results of the Earliest Deadline Scheduling Algorithm," *IEEE Trans. Softw. Eng.*, 1989.

[14] J. Leung and M. Merrill, "A note on preemptive scheduling of periodic, real-time tasks," *Information Processing Letters*, vol. 1980.

[15] A. Burns and A. Wellings, *HRT-HOODTM: A Structured Design Method for Hard Real-Time Ada Systems*, ser. Real-Time Safety Critical Systems, 1995.

[16] I. Lehtoranta, E. Salminen, A. Kulmala, M. Hannikainen, and T. Hamalainen, "A parallel MPEG-4 encoder for FPGA based multiprocessor SoC," in *Int. Conf. on Field Prog. Logic and Apps.*, 2005.

[17] A. Bogdanov, L. Knudsen, G. Leander, C. Paar, A. Poschmann, M. Robshaw, Y. Seurin, and C. Vikkelsoe, "PRESENT: An Ultra-Lightweight Block Cipher," in *Crypt. Hardware and Embedded Systems*, 2007.