Scrubbing Mechanism for Heterogeneous Applications in Reconfigurable Devices

RUI SANTOS and SHYAMSUNDAR VENKATARAMAN, National University of Singapore
AKASH KUMAR, Technische Universität Dresden

Commercial off-the-shelf (COTS) reconfigurable devices have been recognized as one of the most suitable processing devices to be applied in nano-satellites, since they can satisfy and combine their most important requirements, namely processing performance, reconfigurability, and low cost. However, COTS reconfigurable devices, in particular Static-RAM Field Programmable Gate Arrays, can be affected by cosmic radiation, compromising the overall nano-satellite reliability. Scrubbing has been proposed as a mechanism to repair faults in configuration memory. However, the current scrubbing mechanisms are predominantly static, unable to adapt to heterogeneous applications and their runtime variations. In this article, a dynamically adaptive scrubbing mechanism is proposed. Through a window-based scrubbing scheduling, this mechanism adapts the scrubbing process to heterogeneous applications (composed of periodic/sporadic and streaming/DSP (Digital Signal Processing) tasks), as well as their reconfigurations and modifications at runtime. Conducted simulation experiments show the feasibility and the efficiency of the proposed solution in terms of system reliability metric and memory overhead.

CCS Concepts: • **Computer systems organization → Reliability;**

Additional Key Words and Phrases: Fault-tolerance, heterogeneous applications, reconfigurable devices, scrubbing mechanisms

ACM Reference Format:
Rui Santos, Shyamsundar Venkataraman, and Akash Kumar. 2017. Scrubbing mechanism for heterogeneous applications in reconfigurable devices. ACM Trans. Des. Autom. Electron. Syst. 22, 2, Article 33 (February 2017), 26 pages.
DOI: http://dx.doi.org/10.1145/2997646

1. INTRODUCTION

Nano-satellites have been gaining significant importance in space missions, since traditional satellites are expensive to launch. They are lighter (usually under 10kg) and are small in 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 faster development and a lower price due to economy of scale.

This work was supported by the Singapore Ministry of Education Academic Research Fund Tier 1, under the project grant R-263-000-B02-112.
Authors' addresses: R. Santos and S. Venkataraman, Signal Processing and VLSI Lab, Department of Electrical and Computer Engineering, National University of Singapore, 4 Engineering Dr 3, Singapore 117583; emails: {elergvds, shyam}@nus.edu.sg; A. Kumar, Center for Advancing Electronics Dresden, Technische Universität Dresden, 01062 Dresden, Germany; email: akash.kumar@tu-dresden.de.

DOI: http://dx.doi.org/10.1145/2997646

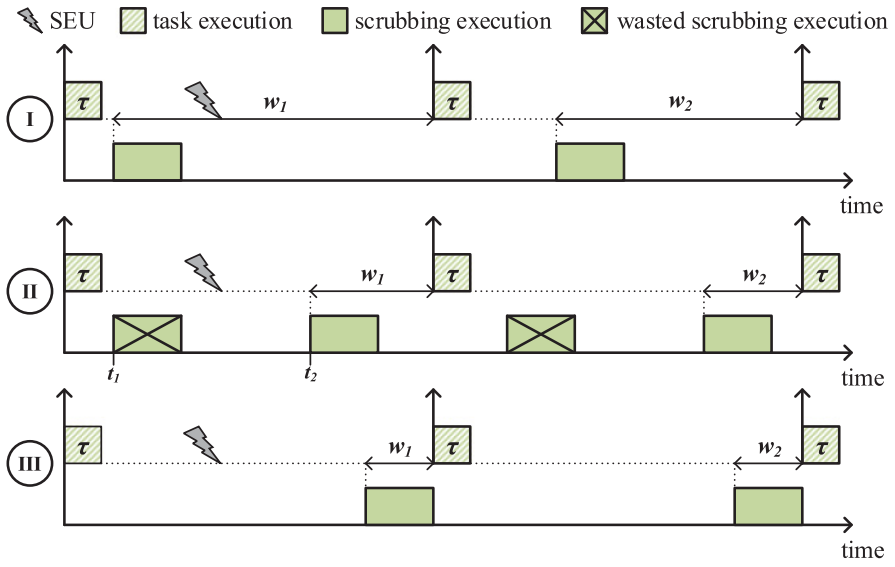

Fig. 1. Motivation example.

Following this trend, COTS Static-RAM (SRAM)-based Field Programmable Gate Arrays (FPGAs) have been significantly used in space environments [Bolchini and Sandionigi 2014; Siegle et al. 2015], 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 [Neuberger et al. 2003; Jing et al. 2013]. These events, which are called Single Event Upsets (SEUs) [Carmichael et al. 2000; Heiner et al. 2009], can inadvertently change the device outputs and corrupt the function results. Hardened FPGAs [Lattice Semiconductor 2015] are one way to mitigate this problem but are very costly. Due to this high price, several fault-tolerance mechanisms have been developed in order to increase the reliability of SRAM-based FPGAs. The most common techniques exploit spatial/hardware redundancy [Cheatham et al. 2006; Koren and Krishna 2007] such as Triple Modular Redundancy (TMR) [Pratt et al. 2006; Bolchini et al. 2007] and Duplication With Compare (DWC) [Johnson et al. 2008; Sarkar et al. 2009]. Besides TMR and DWC, scrubbing is often used to correct errors after the FPGA has been subjected to SEUs. This method takes advantage of the FPGA reconfiguration capability and periodically rewrites the configuration frames of the FPGA, overwriting possible faulty bits caused by SEUs.

A common drawback among the current scrubbing solutions is the independence of the scrubbing execution and the user tasks implemented in 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, please consider the simple example described in Figure 1. A task τ with a periodic behavior is implemented in an FPGA device, using a certain amount of hardware resources. These resources are scrubbed periodically as described in 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 t_2). 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' executions.

Nowadays, general embedded system applications and even nano-satellite applications are increasingly composed of a large number of heterogeneous and dynamic tasks. They may be created and removed from the system during the execution and they may also have different execution behaviors, such as periodic, sporadic, and streaming/DSP (Digital Signal Processing). Moreover, they may also have very different timing and fault-tolerance requirements, which consequently organize them in different levels of criticality or importance for the system [Baruah et al. 2010]. For instance, one fault in a very critical task may have a significant negative impact on the system as compared to multiple faults in a less critical task. Using a decoupled and static scrubbing process from the tasks' execution as the one used by current solutions is not efficient for these heterogeneous embedded systems.

In order to overcome some of these limitations (the decoupled one), the authors of this article have proposed [Santos et al. 2014] a new scrubbing mechanism for user periodic tasks that improves the reliability of the system compared to the existing scrubbing solutions, scheduling the scrubbing process according to the execution and the criticality of each user task. However, this mechanism is also static, that is, it does not efficiently support runtime adaptations and reconfigurations on user task set, jeopardizing the overall system reliability metric. In this sense, a dynamically adaptive scrubbing schedule mechanism was proposed by the same authors [Santos et al. 2015]. This mechanism schedules the scrubbing process based on *windows* at runtime. As a result, this mechanism enables a higher reactive scrubbing process that adapts the scrubbing executions to the user task set reconfigurations and modifications at runtime. Nevertheless, both mechanisms are focused on improving the reliability of periodic and sporadic user application tasks, not considering streaming/DSP applications.

1.1. Contributions

The following items are the key contributions of this article:

—An enhanced scrubbing schedule mechanism that increases the reliability of a heterogeneous user applications, composed of periodic, sporadic, and streaming/DSP applications, implemented in FPGA;
—An improved scrubbing schedule model that allows an efficient and suitable scrubbing process to streaming/DSP applications, modeled by synchronous dataflow graphs (SDFG);
—A window-based scrubbing schedule that allows runtime changes on the scrubbing process, according to changes in the user application;
—A case study that assesses the real applicability of the proposed scrubbing mechanism.

Conducted simulation experiments show the feasibility and the efficiency of the proposed scrubbing mechanism. In particular, they show significant improvements on the system reliability metric, up to 19.2% on average, when compared to other existing scrubbing mechanisms. The experiments also show that the scrubbing process can be very reactive when the scrubbing schedule is performed in *windows*. Moreover, the implementation of these *windows* leads to significant reductions on the memory overhead with small impact on the system reliability metric.

To the best of the authors' knowledge, this is the first work that proposes an efficient scrubbing mechanism for heterogeneous user applications implemented in FPGA. This

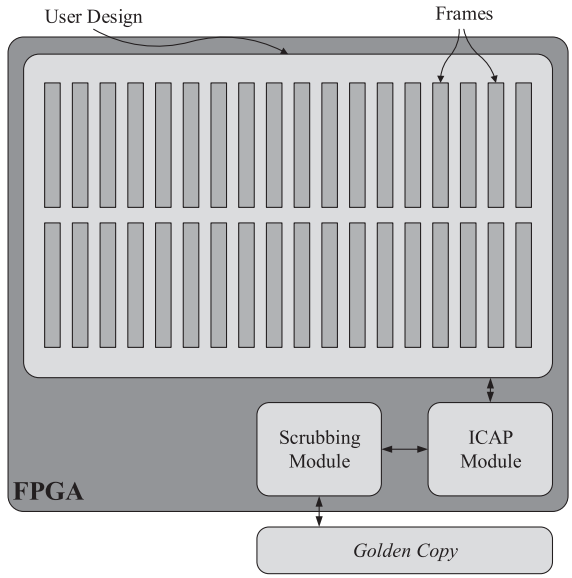

Fig. 2. FPGA – background.

mechanism adapts the scrubbing process to the user applications, according to their execution behavior, criticality, and timing requirements.

1.2. Article Organization

The rest of the article is organized as follows. Section 2 presents the FPGA background and related works concerning the scrubbing mechanisms. Section 3 describes the system model. Section 4 details the problem and deduces its complexity. Section 5 proposes a heuristic to solve it. Section 6 presents a case study. Section 7 introduces the experiments performed and discusses the respective results. Finally, Section 8 presents the conclusions.

2. BACKGROUND AND RELATED WORKS

2.1. FPGA Background

FPGAs are configured using bitstreams, which contain the configuration data for each block of the FPGA such as the Block-RAMS (BRAMs), Look-Up Tables (LUTs), interconnections, and Flip-Flops (FFs) [Xilinx Corporation 2015]. These individual blocks in the FPGA are accessible to the user for building a custom design. Moreover, the FPGA can be seen as a collection of frames, in which each of these individual blocks are implemented, as described in Figure 2. 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 three different ports for reconfiguration [Xilinx Corporation 2012]: JTAG, SelectMAP, and ICAP. In particular, the Internal Configuration Access Port (ICAP) is an internal port that can be accessed within the FPGA for reconfiguration while the other two 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).

It is also important to note that there are other architectures such as the coarse-grained reconfigurable architectures (CGRA) [Liu et al. 2013, 2015] where the proposed scrubbing mechanism can also be applied.

2.2. Scrubbing Related Works

Scrubbing is a mechanism used to repair faults on an FPGA that takes advantage of the FPGA reconfiguration capabilities [Heiner et al. 2009; Guan et al. 2008]. The FPGA reconfiguration is possible through the FPGA ICAP that allows the reading and writing of the FPGA configuration frames, the lowest reconfigurable granular blocks found in an FPGA. Several fault-tolerance solutions have been developed around this mechanism with the simplest approach being blind scrubbing [Carmichael et al. 2000; Brosser et al. 2014; Hoque et al. 2014]. 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. The entire FPGA is scrubbed without considering the tasks that have been implemented in it and the respective used configuration frames. An external memory with continuous access is required to store the original configuration frames, frequently called *golden copy*. Nazar et al. [2013a, 2013b] propose a mechanism that statistically finds the optimal frame to start the scrubbing, which reduces the mean time to repair a certain fault. This mechanism does not scrub the entire FPGA but only the frames that implement the user tasks, that is, the frames that are relevant for the user application. Scrubbing associated with TMR implementation [Hoque et al. 2014; McMurtrey et al. 2008] has also been studied by the researchers. On one hand, this solution allows us to improve the overall system reliability metric when compared to a pure TMR approach. On the other hand, it allows us to reduce the scrubbing frequency and, consequently, the power consumption when compared to a pure scrubbing mechanism. Readback scrubbing is another solution that 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) [Argyrides et al. 2011; Park et al. 2012; Lanuzza et al. 2010; Venkataraman et al. 2014]. This approach enables fault detection by reading the configuration data frame by frame, computing their error correction codes (ECCs) and comparing them to the original ones previously computed and stored externally for each frame. Sari and Psarakis [2011] propose a re-placement method to reduce the number of sensitive frames (frames that are used to implement the user tasks) and thus reduce the scrubbing time. They also combine a scrubbing fault-detection and repair method (based on ECC) with a checkpointing mechanism in order to improve the reliability of the FPGA user design. Sari et al. [2013] also improve this work by proposing a proper analysis that enables the user real-time tasks to meet their deadlines, taking into account the number of checkpoints as well as the required scrubbing time.

All these scrubbing mechanisms are mostly independent of the user applications execution behavior implemented in the FPGA. The FPGA is frequently scrubbed subsequently with a constant and static rate or following a static pattern defined before the execution of the system. As a result, the reliability of the system is not maximized and scrubbing utilization is wasted. In order to solve this drawback, Santos et al. [2014] propose a scrubbing mechanism that improves the efficiency and the reliability of the system. This mechanism schedules the scrubbing process according to the criticality of each user application and also as close as possible to their executions. This way, the probability of the user applications being affected by a fault is reduced. However, this solution is also static, that is, the scrubbing scheduled is computed offline and during the runtime execution any modification/reconfiguration

that may occur on the user application set is not reflected on the scrubbing schedule. The memory required to store the scrubbing schedule may be large, since this solution is dependent on the least common multiple (LCM) among the scrubbing task periods on the system. Moreover, the task model considered in this solution is limited, since it assumes that the user applications are strictly periodic, executing in well-defined instants. As a response, Santos et al. [2015] present a dynamically adaptive scrubbing schedule mechanism for mixed-criticality systems executed on reconfigurable embedded systems, such as FPGAs. This mechanism schedules the scrubbing process based on windows at runtime and following the fixed priority scheduling. As a result, this mechanism enables a higher reactive scrubbing system that adapts the scrubbing executions to the user application set reconfigurations and modifications at runtime. Moreover, this solution significantly reduces the amount of memory required to store the scrubbing schedule. However, neither of the previous works consider an important group of applications that are increasingly found in heterogeneous embedded systems, such as the signal processing (DSP) and multimedia applications.

In this article, the proposed scrubbing mechanism is evaluated comparing it with the other existing scrubbing mechanisms, in terms of reliability achieved by system. It should be noted that the existing approaches can be divided into two classes, *Selective* and *Blind*. Both execute without any strong relation to the user applications execution, penalizing the system reliability metric. As mentioned before, in the literature, the *Blind* scrubbing mechanism [Carmichael et al. 2000; Brosser et al. 2014; Hoque et al. 2014] is described as a simple mechanism that replaces the values of the configuration memory, and for that it uses the initial configuration bit file stored in a golden copy. It replaces the entire configuration memory, even if part of it is not used by the user application. However, please note that the term *Blind* scrubbing is related to the fact that the mechanism is blind to the actual occurrence of an error, not to the fact of being independent of the applications implemented in the FPGA. Because of this aspect, the *Blind* scrubbing terminology used in this article can also be referred as *Full Memory* scrubbing. On the other hand, in this article *Selective* scrubbing only scrubs the configuration frames that are utilized by the user design. For that, information about the FPGA placement of the user applications is required as used in Nazar et al. [2013a, 2013b], for instance.

3. SYSTEM MODEL

3.1. Architecture Model

FPGAs have the ability to integrate on the same device several functionalities running in parallel and implemented in well-defined FPGA regions [Kumar et al. 2008] (Figure 3). The user design of an FPGA can be divided into several regions/partitions, commonly called partially reconfiguration regions (PRR). Each partition is responsible for implementing one or more specific functionalities, 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 very well delimited and, therefore, is composed of a well-defined set of configuration frames. In the context of this work, it is assumed that the diverse user application tasks are implemented by a set of PRRs, one task per region as described in Figure 3.

3.2. Application Model

As mentioned above, embedded system applications designed for aerospace equipments are nowadays becoming more and more complex. Following this trend, FPGA devices

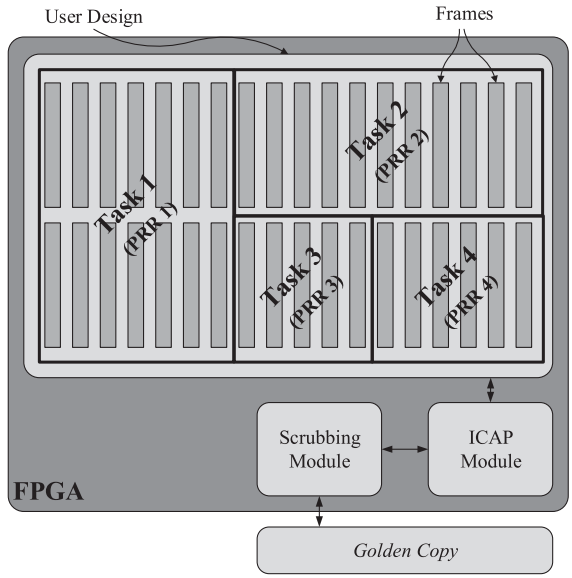

Fig. 3. Architecture model.

have to accommodate diverse heterogeneous applications with different execution behaviors (streaming/DSP and periodic/sporadic), timing, and fault-tolerance requirements. Therefore, the challenge is to create an integrated, efficient and fair scrubbing mechanism that is able to provide fault tolerance to the user applications according to their characteristics. In this work, an embedded system (ES) can be composed of a set of applications $App_a \in ES$. Each application App_a can be identified as either streaming/DSP or periodic/sporadic. Associated to each application there is a criticality value $crt_a \in \mathbb{N}$ that models the importance of this application to the entire system. Zero criticality corresponds to the lowest criticality in the system.

3.2.1. Streaming/DSP Application. A streaming/DSP application ($App_a \in ES$) has been naturally modeled by the SDFG paradigm [Hölzenspies et al. 2008; Cannella et al. 2014]. Considering the dataflow model defined by Kavi et al. [1986], an SDF graph is composed of one or more nodes, called actors. The actors represent application tasks and their execution consists of reading and computing *tokens* (data items) from their input ports and writing the computation results as *tokens* on the output ports. Therefore, an actor is only fired/executed when it has sufficient tokens on its inputs and enough buffer space on the outputs to store the produced tokens.

Definition 3.1 (*SDFG G*). A SDFG G [Ghamarian et al. 2006] can be represented as a directed graph, $G = (V, E)$, composed of a finite set of actors $\tau_i \in V_G$ and a set of directed edges E_G. Each edge $e_u \in E_G$ is defined by a tuple $e_u = (\tau_i, \tau_j)$, denoting the communication from the actor τ_i to the actor τ_j. When τ_i executes, it transmits s_i^u data tokens on the edge e_u. On the other hand, when τ_j executes, it consumes c_j^u data tokens on the edge e_u.

Figure 4 presents an SDFG example. This graph G is composed of four actors $V_G = \{\tau_0, \tau_1, \tau_2, \tau_3\}$ and a set of edges $E_G = \{e_0, e_1, e_2, e_3, e_4\}$. For instance, when the actor τ_1 executes, it consumes $c_1^2 = 1$ data tokens on the edge e_2 and then transmits $s_1^3 = 1$ data tokens on the edge e_3.

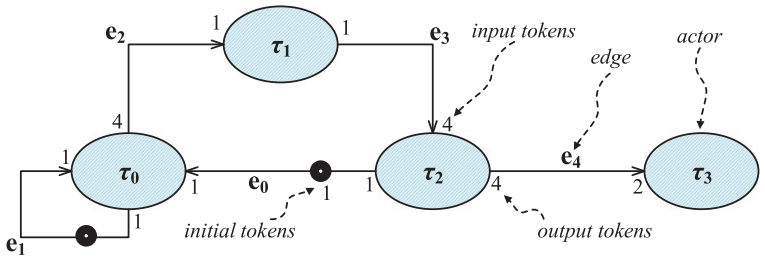

Fig. 4. SDFG G example.

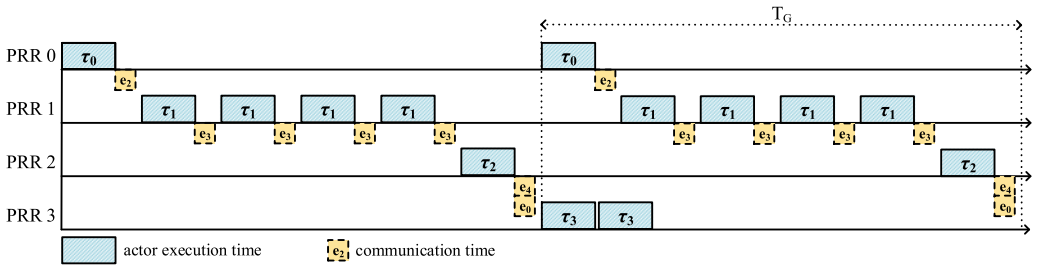

Fig. 5. A feasible schedule S_G for the SDFG G.

Definition 3.2 (*SDFG Schedule, S_G*). A schedule S_G of an SDFG G represents the order of execution (firings) of all actors that are part of the graph G. A schedule is called feasible if the sequence of the actors' execution guarantees an SDFG execution free of deadlocks and buffer overflow conditions. A feasible schedule can be repeatedly executed.

Definition 3.3 (*SDFG Iteration Period, T_G*). An iteration period T_G of an SDFG G represents time between the execution of consecutive iterations of schedule S_G.

Definition 3.4 (*Actor Execution Pattern, EP_i*). The actor execution pattern (EP_i) defines for the actor $\tau_i \in V_G$ the pattern of firings during the iteration period T_G. $EP_i(l)$ defines the actor execution instance l inside the iteration period, where $0 <= l < |EP_i|$ and $|EP_i|$ defines the number of execution instances of the actor τ_i during the iteration period T_G.

Taking into account the target architecture model described in the previous subsection, each actor is mapped in an independent processing area (FPGA partition). Therefore, actors execute by following a self-timed schedule fashion, that is, they execute as soon as they get the sufficient number of tokens on their input edges. Considering this approach, Figure 5 presents one feasible SDFG schedule of the SDFG described in Figure 4. Please note that during an iteration period T_G the application actors τ_0, τ_1, τ_2, and τ_3 execute 1, 4, 1, and 2 times, respectively.

As noted, each actor of each SDFG G executed on the FPGA is implemented in an independent FPGA partition. Therefore, each actor τ_i implemented in the FPGA, with $0 <= i < \sum_{\forall G} |V_G|$, can be characterized by the following parameters that will be useful to define the scrubbing process, $\tau_i = (C_i, T_i, EP_i, \eta_i, \zeta_i)$. C_i defines the execution time of the actor τ_i; T_i represents the actor execution pattern period, which is equal to the respective graph iteration period T_G; EP_i defines the actor execution pattern inside the iteration period; η_i defines the number of FPGA configuration frames used to implement the actor τ_i on the FPGA; and, finally, ζ_i represents the criticality of the

actor in the system, restricted by the criticality assumed by the SDFG application to which it belongs (this relation will be detailed later, in subsection 3.5). The execution of the actor τ_i can be interpreted as following: τ_i is released inside the iteration period according to the execution pattern EP_i and this pattern is executed in an infinite sequence at the instants kT_i, where $k = \{0, 1, 2, 3, \dots\}$.

3.2.2. Periodic/Sporadic Application. Without loss of generality, the embedded system can also include periodic/sporadic applications ($App_a \in ES$) as elaborated in Santos et al. [2014] and Santos et al. [2015]. Each periodic/sporadic application App_a is naturally associated to a task τ_i characterized by the same parameters as the ones that characterize an SDFG actor ($\tau_i = (C_i, T_i, EP_i, \eta_i, \zeta_i)$). Therefore C_i indicates the task execution time; T_i represents the task's period; EP_i is composed of only one entry, since the task is only executed once per period (so $EP_i(0)$ represents the task offset); η_i also defines the number of FPGA configuration frames used to implement the task; and, finally, ζ_i represents the criticality directly inherited by the application's criticality, crt_a.

3.2.3. User Tasks—Uniformalization. Since the actors that compose an SDFG application and the periodic/sporadic tasks can be formally defined with the same parameters, in the rest of the article, the term **user tasks** will be generally used to refer to both. Therefore, Γ will be used to represent the set of user tasks τ_i implemented in the system.

3.3. Error Model

As mentioned on Section 2, the FPGA device composed of Θ configuration frames can be affected by SEUs. The SEUs follow a Poisson distribution with a rate of λ failures per time unit [Bridgford et al. 2008]. Therefore, the probability of no failures in an unprotected FPGA-based design in an interval t is given by the following equation [Das et al. 2013; Axer et al. 2011]:

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

Definition 3.5 (*Reliability of an Application*). Reliability of an application App_a is a metric that defines the probability of App_a being executed in the interval $[0, t]$ without faults.

Therefore, the reliability of App_a can be expressed in the following equation as the probability of all instances of all associated user tasks τ_i being executed without faults in the interval $[0, t]$:

$$R_{App_a}(t) = \prod_{\tau_i \in App_a} \prod_{k=0}^{\lfloor t/T_i \rfloor} P_{ne}\left[\tau_i^k\right]. \tag{2}$$

The previous equation only computes the reliability from 0 to $\lfloor t/T_i \rfloor \times T_i$. The reliability of the remaining time is negligible, since t is significantly higher than T_i. $P_{ne}\left[\tau_i^k\right]$ represents the probability of the user task execution instances (τ_i) on the iteration period k being executed without any fault in all its frames f during the interval $w_i^{k,f}$. Therefore, $P_{ne}[\tau_i^k]$ is defined as follows:

$$P_{ne}\left[\tau_i^k\right] = \prod_{f=0}^{\eta_i - 1} e^{-\frac{\lambda}{\Theta} w_i^{k,f}}. \tag{3}$$

Regarding frame f, $w_i^{k,f}$ defines the time interval between the end of the last user task instance $\tau_i(|EP_i| - 1)$ on the iteration period k and the closest of two points (previous

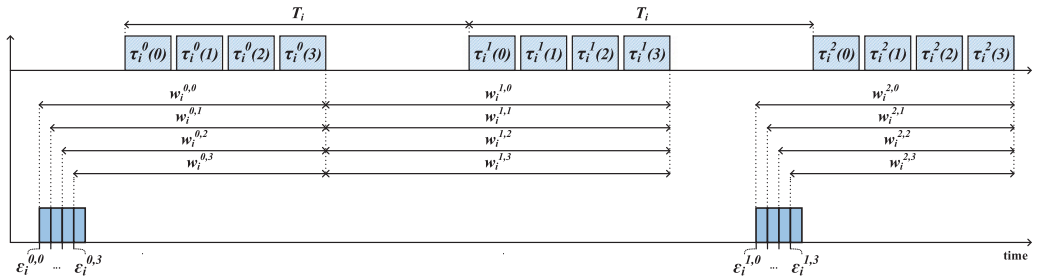

Fig. 6. Time interval between last scrubbing process, in particular the frame f, and the beginning of the l^{th} user task execution on the iteration periods 0 and 1.

scrubbing instance of frame f or the end of the last $\tau_i(|EP_i|-1)$ instance on the iteration period $k-1$), as illustrated in Figure 6. If the scrubbing execution of frame f occurs in the middle of a τ_i execution sequence, then each execution instance can be considered individually, improving the application reliability slightly.

Definition 3.6 (*System Reliability Metric*). In this article, system reliability metric is a metric that defines the reliability of the overall system during the interval $[0, t]$ and taking into account the reliability as well as the criticality of each application App_a executing in the system, either SDFG or periodic/sporadic.

Therefore, the system reliability metric can be expressed as follows:

$$R(t) = \sum_{App_a=0}^{|ES|-1} R_{App_a}(t) \times crt_a, \tag{4}$$

where $|ES|$ is the number of applications that are executing in the embedded system (FPGA).

3.4. Scrubbing Model

The proposed scrubbing mechanism does not scrub the FPGA frames sequentially with a constant rate; instead, it scrubs the FPGA configuration frames associated with each user task τ_i independently from others, enabling adaptive scrubbing. The FPGA configuration frames associated with the user task τ_i are scrubbed, taking into account its execution pattern as well as its criticality. In the scrubbing mechanism proposed in Santos et al. [2014] specially designed for periodic tasks, the scrubbing process executes periodically and just before (as near as possible) the user tasks execution. There is a one-to-one relationship between the periodic user tasks and the scrubbing process, that is, associated with each user task there is only one scrubbing task. When this solution is used on streaming/DSP applications modeled by SDFGs, one of the possibilities is to execute the scrubbing tasks periodically and just before the first actor τ_i execution instance in each iteration period. However, this solution may not be the most reliable one. Please consider the example in Figure 7 in order to better understand this point. In SDFGs, an actor can execute several times in one iteration period, like actor τ_1 (Figure 7(I)). If the corresponding scrubbing task only executes once in one iteration period, for instance, just before the execution of the first actor instance (Figure 7(II)), then it may not be enough to keep a desired level of reliability. The other actor execution instances ($\tau_1(1)$, $\tau_1(2)$, $\tau_1(3)$) are increasingly far from the last corresponding scrubbing task execution instance ($s\tau_1$), increasing the probability of them being executed with a fault. This effect is amplified with the increase in the number of actor instances per iteration period. In order to better protect the SDFGs

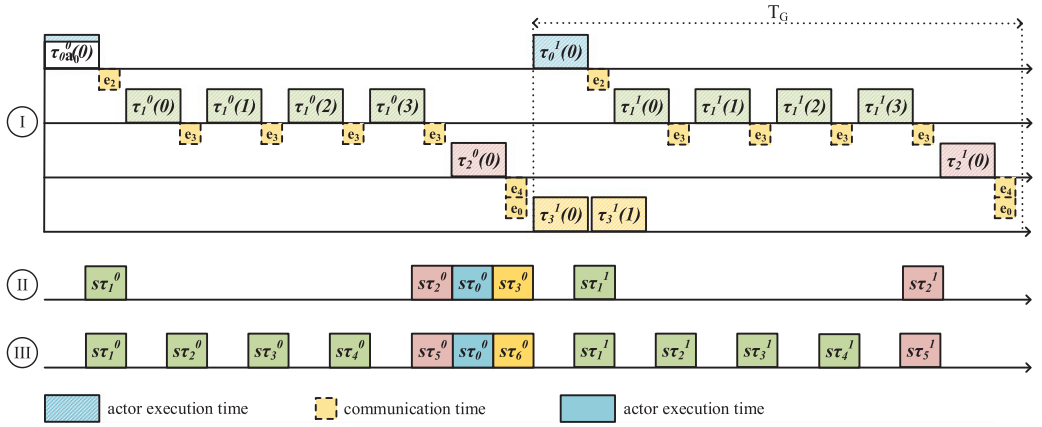

Fig. 7. Scrubbing execution. (I) User task execution example. (II) Scrubbing execution: Only one scrubbing task associated with each user task. (II) Scrubbing execution: More than one scrubbing task can be associated with one user task according to their execution behavior.

implemented in the FPGA system, each actor τ_i can be associated with one or more scrubbing tasks ($s\Gamma_i$, the scrubbing task set associated with actor τ_i). In the ideal case, each actor execution instance ($EP_i(l)$) is associated with its own scrubbing task, as described in Figure 7(III). However, this scenario may not be scalable and practical, since a huge number of all scrubbing tasks depending on the SDFG could be generated. Therefore, a balanced solution needs to be found.

Before presenting one possible solution for this problem, please take into account first the formal definition of the scrubbing process. The scrubbing process can be defined by the union of all the scrubbing task sets $\{s\Gamma_0 \cup s\Gamma_2 \cup ... \cup s\Gamma_i\}$ associated with each user task τ_i, with $0 <= i < |\Gamma| - 1$ and where $|\Gamma|$ represents the number of user tasks implemented in the FPGA. Each scrubbing task $s\tau_j \in \{s\Gamma_0 \cup s\Gamma_2 \cup ... \cup s\Gamma_{|\Gamma|-1}\}$, with $0 <= j < |s\Gamma_0 \cup s\Gamma_2 \cup ... \cup s\Gamma_{|\Gamma|-1}|$, is straightforwardly modeled as a periodic task characterized by five parameters $s\tau_j = (\tau_i, SC_j, ST_j, \Phi_j, \zeta_j)$: τ_i is the user task, with which the scrubbing task $s\tau_j$ is associated; SC_j defines the time to scrub the η_i FPGA frames used to implement the user task τ_i; ST_j represents the period of this scrubbing task; Φ_j is the initial offset; and ζ_j defines the criticality inherited from user task τ_i. The execution of $s\tau_j$ can be interpreted as follows: $s\tau_j$ is released in an infinite sequence of jobs at the instants $\Phi_j + pST_j$, with $p = \{0, 1, 2, 3, \dots\}$. Note that the release of the scrubbing jobs are synchronized with the corresponding user task jobs execution. Considering the deadline of $s\tau_j$ to be equal to its period ST_j, the job instance $s\tau_j^p$ has to execute SC_j time units during the interval $[\Phi_j + pST_j, \Phi_j + (p+1)ST_j)$.

Algorithm 1 describes one possible solution to define the scrubbing task set associated with each user task. This algorithm receives as input the implemented user task set (Γ) and, consequently, the execution pattern EP_i of each user task τ_i. It also receives the maximum acceptable time distance (Υ) between the user task execution instance and last corresponding scrubbing execution. At the end, the algorithm returns the scrubbing task set $s\Gamma_i$ with the respective parameters of each scrubbing task $s\tau$. For all the user tasks τ_i (line 1) and for each execution instance $\tau_i(l)$ (line 6) during the period T_i, the algorithm verifies the time distance between the current user task execution instance and the last corresponding scrubbing execution (line 7). Please assume that the first scrubbing execution occurs just before the execution of the first user task execution instance (lines 2 to 5). If this distance is greater than Υ (line 7), then a new

ALGORITHM 1: Finding the scrubbing task set $s\Gamma_i$ associated with each user task τ_i.

Input: Γ, Υ;
Output: $s\Gamma_0 ... s\Gamma_{|\Gamma|-1}$;
1: **for** (i = 0; i < $|\Gamma|$; i ++) **do**
2: $lastS = EP_i(0)$;
3: $s\tau \rightarrow \tau_i = i$; $s\tau \rightarrow SC = \eta_i \times$ *time to scrub one frame*; $s\tau \rightarrow \zeta = \zeta_i$;
4: $s\tau \rightarrow \Phi = EP_i(0) - (s\tau \rightarrow ST)$;
5: $s\Gamma_i = s\Gamma_i \cup s\tau$;
6: **for** ($l = 1; l < |EP_i|; l++$) **do**
7: **if** $((EP_i(l) - lastS) > \Upsilon)$ **then**
8: $lastS = EP_i(l)$;
9: $s\tau \rightarrow \tau_i = i$; $s\tau \rightarrow SC = \eta_i \times$ *time to scrub one frame*; $s\tau \rightarrow \zeta = \zeta_i$;
10: $s\tau \rightarrow \Phi = EP_i(l) - (s\tau \rightarrow ST)$;
11: $s\Gamma_i = s\Gamma_i \cup s\tau$;
12: **end if**
13: **end for**
14: **end for**
15: **return** $s\Gamma_0 .. s\Gamma_{|\Gamma|-1}$;

scrubbing task $s\tau_j$ is created and associated with the task τ_i (lines 8 to 11). Its offset Φ_j will be equal to the execution instance of the user task execution instance $\tau_i(l)$.

Please note that a scrubbing preemptive approach is considered in this article. Therefore, in order to guarantee that no scrubbing preemptions occur during the scrubbing process of one frame, the minimum time unit used to define the scrubbing schedule is equal to the time to scrub one FPGA frame. Therefore, the periods (T) and, consequently, the corresponding scrubbing periods (ST) have to be multiples of the time to scrub one configuration frame. The minimum time unit imposed by this assumption is not a limiting factor. In a real system, this time is very small. It usually takes only a few microseconds (μs). For instance, considering a Xilinx Virtex-6 FPGA (XC6VLX240T), the time to scrub one frame at the maximum ICAP frequency is 0.81μs.

3.5. Criticality Assignment

Definition 3.7 (*Criticality of an Application*). Criticality of an application is a metric that defines the impact of its correct (or incorrect) functioning on the overall system correctness [Dobrin et al. 2008].

As defined, the criticality of an application reflects the importance/impact of that application on the system. In this sense, the application's criticality plays a central role, when the scrubbing task sets have to be defined, since the ICAP capacity is limited and it may not be enough to scrub all the user tasks associated to the applications at the maximum frequency desired. In this case, the tasks associated to the most critical applications must be scrubbed more frequently than the less critical ones. The following method is proposed to assign the application's criticality. However, on this aspect the proposed scrubbing mechanism is very flexible. Therefore, different criticality assignment methods may be used.

3.5.1. Criticality Based on the User Criteria—Cu. This method assigns the criticality according to the user option. Therefore, each application assumes the criticality given by the following equation:

$$\forall App_a \in ES, crt_a = norm_1(constant_a), \tag{5}$$

where $constant_a \in \mathbb{N}$ is the value defined by the user and $norm_1()$ gives the normalized value by 1, that is,

$$norm_1(constant_a) = \frac{constant_a}{\sum_{a=0}^{|ES|-1} constant_a}. \tag{6}$$

Regarding the user tasks, they will inherit the criticality of the application that they belong to. Therefore,

$$\forall \tau_i \in \Gamma, \tau_i \in App_a, \zeta_i = \frac{crt_a}{|App_a|}, \tag{7}$$

where $|App_a|$ represents the number of user tasks associated to the application App_a. Please note that the assignment of the criticality to the different applications in the system and also to their tasks is completely flexible. The user can utilize different equations according to his/her preference.

4. SCRUBBING SCHEDULE—PROBLEM DEFINITION

After defining the scrubbing process, that is, after defining the scrubbing task set $s\Gamma_i$ associated with each user task τ_i and its executions instances, the ideal case that maximizes the reliability of each task τ_i during an interval $[0, t]$ is to scrub the corresponding FPGA frames with a period equal to the user task period (T) and as close as possible before their execution instances, as described in Figure 7. However, this ideal case may not be possible to achieve, since the ICAP resources/utilization (time to read and write the user design frames) are limited. In that case, the scrubbing task periods (ST) may have to be larger than the corresponding user task periods T. Taking into account these concerns, the problem can be formulated as follows. Given all user tasks τ_i implemented in the FPGA using a certain number of configuration frames η_i, the challenge is to determine the exact schedule of all scrubbing tasks $s\tau_j$ in a way that it enables the fault-tolerance in all user tasks in the system and reduces the probability of each user task execution instance $\tau_i^k(l)$ being affected by a fault. The exact scheduling means the determination of the scrubbing instant of each frame in each scrubbing instance $\varepsilon_j^{p,f}$ (refer to Figure 6). The most critical user tasks in the system must be the most reliable, that is, they must be scrubbed more frequently and as close as possible to their execution instants. On the other hand, the less critical user tasks must be the least reliable if there are not enough scrubbing resources to scrub all the user tasks in the system. In this case, the less critical tasks must be scrubbed less frequently. In short, the global objective is to maximize the system reliability metric based on the application criticality. Therefore, taking Equations (2) and (4) into account, the objective function can be expressed as follows:

$$\mathbf{max}\, R(t) = \mathbf{max} \sum_{App_a}^{|ES|-1} \left(\prod_{\tau_i \in App_a} \prod_{k=0}^{\lfloor t/T_i \rfloor} P_{ne}\left[\tau_i^k\right] \right) \times crt_a, \tag{8}$$

where $|ES|$ returns the number of user applications implemented in the system and $\lfloor t/T_i \rfloor$ gives the number of user task periods in a pre-defined time interval $[0, t]$.

4.1. Complexity

The scrubbing problem, that is, finding the exact execution instant for each scrubbing task that maximizes the system reliability metric, can be modeled as a well-known preemptive scheduling problem. However, these problems are NP-hard in the strong sense as Jeffay et al. [1991] have shown. In the next section, a heuristic is proposed in order to find a feasible solution in a suitable interval of time.

5. PROPOSED HEURISTIC

Considering Figure 6, as one can observe, if the scrubbing tasks are scheduled as close as possible to their deadlines, the probability of the corresponding user tasks being affected by a fault is reduced. Therefore, instead of finding the exact schedule 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 [Chetto and Chetto 1989] is used to schedule the scrubbing tasks, since it schedules the tasks as late as possible near the deadlines.

ALGORITHM 2: Proposed Heuristic (preemptive)

Input: Γ, $s\Gamma$;
Output: schedule;
1: findSPeriods(Γ, $s\Gamma$, uBound); *//using ILP*
2: lcm = computeLCM($s\Gamma$);
3: schedule = edlSchedule($s\Gamma$, lcm);
4: **return** schedule;

Algorithm 2 describes the proposed heuristic in order to compute the scrubbing schedule. Three main steps can be highlighted. The first step that is implemented by the function *findSPeriods* (line 1) finds the minimum period of the scrubbing task $s\tau_j$ associated with the corresponding application actor τ_i. This step is performed by using an Integer Linear Programming (ILP) formulation. Equation (9) describes the cost function,

$$\textbf{min} \sum_{j=0}^{|s\Gamma_0 \cup \ldots \cup s\Gamma_{|\Gamma|-1}|-1} \frac{ST_j}{T_i} \times \zeta_i, s\tau_j \in s\Gamma_i, 0 \leq i \leq |\Gamma| - 1. \tag{9}$$

Please note that the scrubbing periods are found according to the user task criticality. These periods must make all scrubbing task sets $s\Gamma_i$ schedulable, verifying the utilization-based schedulability condition given by the following equation:

$$\sum_{j=0}^{|s\Gamma_0 \cup \ldots \cup s\Gamma_{|\Gamma|-1}|-1} \frac{SC_j}{ST_j} \leq uBound, \tag{10}$$

where $uBound$ defines the maximum ICAP utilization provided to the scrubbing mechanism and $|\Gamma|$ is the number of user tasks implemented in the system. The second step, implemented by the function *computeLCM* (line 2), computes the LCM of the obtained periods in the previous step. The schedule produced in the next step has a cyclic property, that is, for every LCM interval, the schedule is repeated [Leung and Merrill 1980]. Moreover, its feasibility is automatically assured for the LCM interval by the first step (line 1), since the EDL algorithm is optimal for preemptive task sets. The third step, which is implemented by the function *edlSchedule* (line 3), computes the system scrubbing schedule for the LCM interval following the EDL algorithm [Chetto and Chetto 1989].

The memory required to store the scrubbing schedule may be large, since this solution is dependent on the LCM among the scrubbing task periods on the system. Moreover, this dependency of the scrubbing schedule on the LCM interval does not allow the modifications/reconfiguration on the user task set, such as adding, removing,

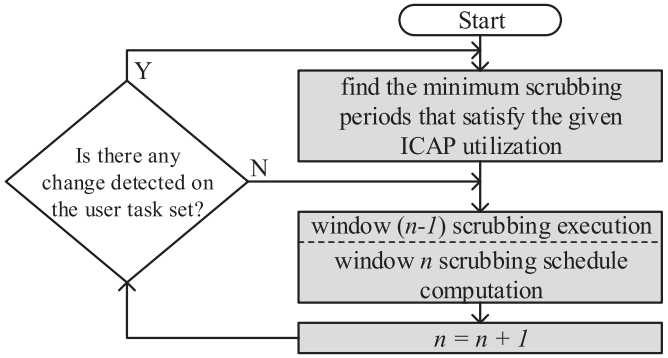

Fig. 8. Scrubbing schedule window mechanism—flowchart.

and modifying tasks. In order to overcome this limitation, a window-based scrubbing schedule mechanism is presented in the next subsection.

5.1. Window-Based Scrubbing Schedule

The scrubbing mechanism of an FPGA device can be classified as a soft real-time system. If some user task is not scrubbed at the right instant according to the computed scrubbing schedule, then there is no negative impact on its execution, that is, the user task is executed normally. However, the reliability of the system may decrease, that is, the probability of that user task to be executed with a fault may increase. Taking this factor into account, the proposed mechanism computes the scrubbing schedule in pre-defined time windows, enabling reactive adaptations of the scrubbing process to the changes on the implemented user task set. Scheduling the scrubbing tasks based on windows is not the optimal solution, since the scrubbing tasks may miss their deadlines. However, the reliability of the system is improved through a reactive update of the scrubbing mechanism to the changes and the execution of the implemented user tasks.

The scrubbing schedule windows have duration Δ (defined as the multiple of the time to scrub one configuration frame) specified by the user and according to the reactivity requirements of the scrubbing process. Their execution can be interpreted as follows: The scrubbing schedule windows are executed sequentially; each window starts at the instants $n \times \Delta$ with $n = \{0, 1, 2, 3, \dots\}$ and has a duration of Δ. Therefore, the window n defines the scrubbing schedule for the interval $(n\Delta, (n+1)\Delta]$. Figure 8 describes the sequence of steps regarding the proposed approach. The first step computes the minimum scrubbing periods according to the criticality of each user task (Equation (9)) that maximize the allowed ICAP utilization defined for the scrubbing mechanism (Equation (10)). Then the scrubbing schedule is executed and computed in windows. During the execution of the scrubbing schedule defined in the window $n-1$, the scrubbing schedule for the window n is computed. Only the scrubbing task activations that occur in the interval $(n\Delta, (n+1)\Delta]$ will affect the schedule produced by the window n. If during the scrubbing schedule execution of the window n any change on the user task set is detected, then the minimum scrubbing periods that satisfy the maximum defined ICAP utilization have to be recomputed. The new periods are then used to compute the scrubbing schedule for the incoming windows. The scrubbing schedule is computed according to the fixed priority scheduling (FPS), contrary to the solution proposed by Santos et al. [2014]. FPS is more predictable and easier to implement than dynamic priority scheduling (DPS) and the definition of criticality fits perfectly on the notion of priority, too. The most critical user tasks are always scrubbed closer

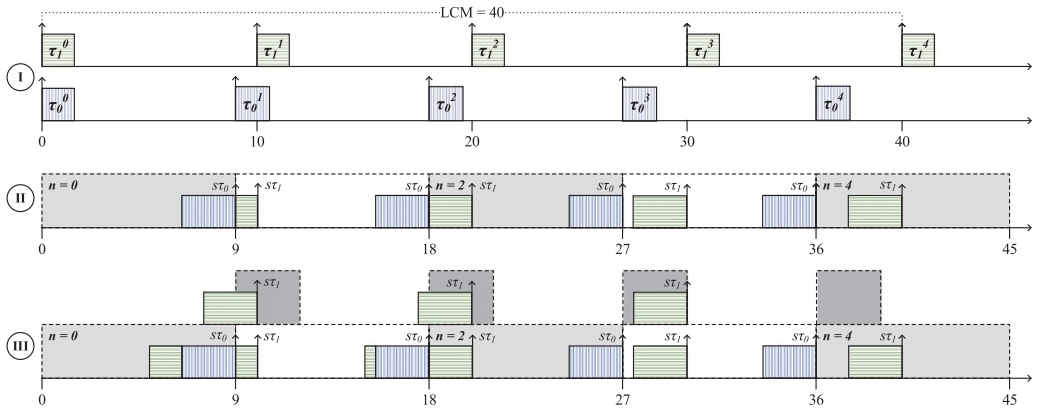

Fig. 9. Auxiliary Window – Example. (I) User task execution. Task τ_0 is more critical than task τ_1. (II) Computed scrubbing schedule using the proposed scrubbing schedule based on windows but without using an auxiliary schedule window. (III) Computed scrubbing schedule using the proposed scrubbing schedule based on windows and improved by using the auxiliary schedule window mechanism. Note 1: The auxiliary windows are represented by the darker boxes. The tasks inside them are the tasks considered for the scrubbing schedule of the previous window. Note 2: The execution time of the user tasks may not be in scale with the scrubbing executions.

to their executions instances than the less critical ones. The better schedulability of the DPS is almost irrelevant in this context, since the scrubbing mechanism should be used for low-ICAP utilizations, leaving free space for FPGA reconfigurations. Note that the memory required by this approach is only the memory required to store the scrubbing schedule of two Δ windows. The gains in terms of memory will be really high when the proposed approach uses small windows sizes (Δ) and the LCM interval given by the task periods is really large. Moreover, if the criticality of the applications has not changed, the scrubbing schedule has to be computed for all windows, since the window size may not be equal to the LCM of all applications periods. However, if the window size happens to be a multiple of the LCM, then it is only needed to compute the scrubbing schedule during changes. This way, any change on the application set can be easily accommodated on the scrubbing schedule.

5.1.1. Auxiliary Window—Improvement. The scrubbing schedule based on windows raises a problem. The scrubbing tasks that are activated in the beginning of the window n may not be executed since the remaining execution space in window n may not be enough. In order to solve this limitation, an auxiliary scrubbing schedule window is proposed. Every scrubbing schedule window n is succeeded by an auxiliary window with Ω size (multiple of the time to scrub one configuration frame), which helps to compute the scrubbing schedule. This way, for the produced scrubbing schedule in the window n, all the scrubbing requests in the interval $(n\Delta, (n+1)\Delta + \Omega]$ are considered. However, the produced schedule in the auxiliary window $((n+1)\Delta, (n+1)\Delta + \Omega]$ is not taken into account, since it will be recomputed in the scrubbing schedule window $(n+1)$. Please observe Figure 9 in order to better understand the auxiliary window mechanism. Diagrams (II) and (III) show the scrubbing schedule for the two user applications presented on diagram (I). The scrubbing schedule window has duration of 9 time units and the auxiliary window of 3. Diagram (II) shows the scrubbing schedule without using the auxiliary window mechanism, unlike diagram (III). Please consider the activation of scrubbing task $s\tau_1$ in window 1 (Figure 9(II)). In this case, window 1 does not have enough space to execute completely the scrubbing task $s\tau_1$. The scrubbing part that was not executed should have been done in window 0. However, this did not happen, since the activation of $s\tau_1$

Table I. User Tasks—Γ (Parameters)

τ_i	User Task Name	*Application*	$C_i(ms)$	$T_i(ms)$	η_i	$\zeta_i(cu)$
τ_0	*Control_Law*	App_0 / *Periodic*	0.900	50.000	250	0.250
τ_1	*Process_IRES_Data*	App_1 / *Periodic*	0.410	100.00	150	0.250
τ_2	*Calibrate_Gyro*	App_2 / *Periodic*	0.390	100.00	100	0.250
τ_3	*Motion_Estimation*	App_3 / *SDFG*	1.910	10.345	1000	0.050
τ_4	*MB_Encoding*	App_3 / *SDFG*	0.084	10.345	42	0.050
τ_5	*MB_Decoding*	App_3 / *SDFG*	0.062	10.345	31	0.050
τ_6	*VLC*	App_3 / *SDFG*	0.130	10.345	65	0.050
τ_7	*Motion_Compensation*	App_3 / *SDFG*	0.057	10.345	26	0.050

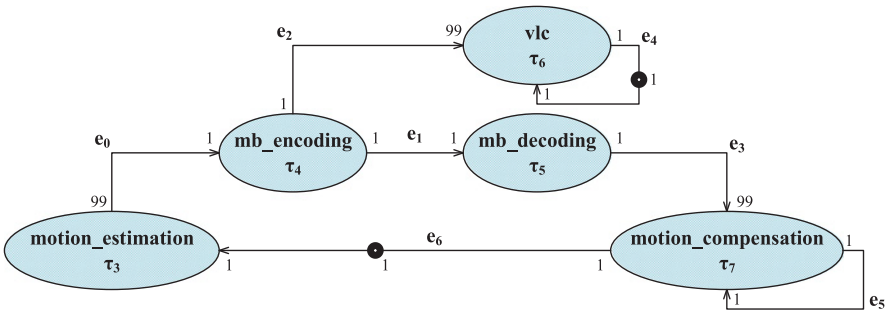

Fig. 10. *h.263* encoder SDFG.

only occurs in window 1. This problem can be solved if, during the computation of the scrubbing schedule of window 0, the scrubbing activation that takes place in the beginning of window 1 is also considered. Therefore, during the computation of the scrubbing schedule of window n, all the scrubbing activations that occur during the auxiliary window placed in the beginning of the window $n+1$ are taken into account (Figure 9(III)).

6. CASE STUDY

A nano-satellite is generally composed of two subsystems, 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 in this case study aims to collect high-resolution video images from the Earth. For that, the implementation of eight user tasks in an SRAM-based FPGA was simulated, whose parameters are defined in Table I. The first three user tasks, based on Forget et al. [2010] and Burns and Wellings [1995], are related to the nano-satellite navigation subsystem, and they present a periodic behavior. The other five tasks are related to the payload subsystem. They implement the *h.263* encoder application, which is responsible for encoding the video frames captured by the nano-satellite camera to be transmitted to the Earth. The *h.263* encoder is modeled by a SDFG, as described in Figure 10. Please note that each of the user tasks is implemented with dedicated hardware in independent FPGA areas (PRRs). In this sense, the execution of all user tasks is never delayed by the lack of execution resources. Figure 11 shows the *h.263* SDFG execution (schedule). During one iteration period, *MB_Encoding* and *MB_Decoding* execute 99 times per period. Contrarily, *Motion_Estimation*, *Motion_Compensation*, and *VLC* execute only once.

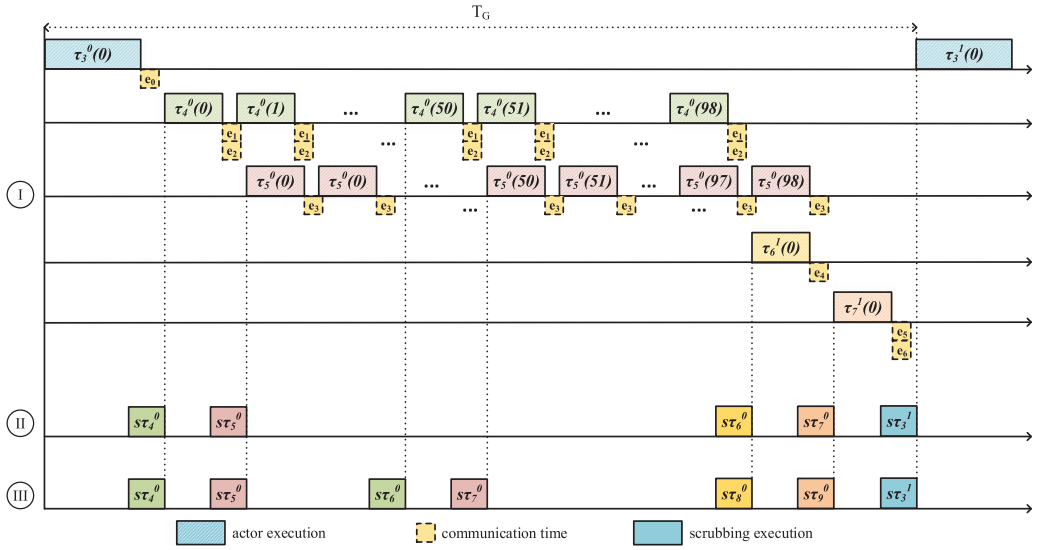

Fig. 11. (I) *h.263* encoder SDFG schedule. (II) Proposed scrubbing execution for $\Upsilon = 11$ms. (III) Proposed scrubbing execution for $\Upsilon = 4.13$ms.

Table I also shows the number of FPGA configuration frames (η_i)[1] used to implement each one of the user tasks. The SRAM-based FPGA (Virtex-6 LX240T) considered in this case study contains 28,464 configuration frames, and it is assumed that each frame of the FPGA device is scrubbed at the maximum ICAP frequency (100MHz). Therefore, each FPGA configuration frame requires 0.81μs to be scrubbed. In order to reduce the energy consumption, the maximum percentage of ICAP utilization provided to the scrubbing mechanism was limited to 30%. Moreover, for this case study, the FPGA device was simulated to be placed in the space environment, subjected to SEUs with a rate $\lambda = \frac{1}{1Hour}$ [Bridgford et al. 2008]. Table I shows as well the criticality assigned to the user tasks regarding the method presented in Section 3.5. The method *cu* assigns the criticality according to the user criteria. In this particular case, it is assumed that all the applications have the same criticality/importance for the system (0.25—normalized). Please note that the user tasks that belong to the SDFG application receive its criticality divided equally among them.

According to the proposed scrubbing solution, the first step defines scrubbing task set $s\Gamma_i$ associated with each user task τ_i. Each scrubbing task set $s\Gamma_i$ is computed based on Algorithm 1. The number of scrubbing tasks $|s\Gamma_i|$ associated with each task τ_i is defined by the maximum time distance (Υ) allowed between the task execution instance and the last scrubbing execution. Therefore, the number of scrubbing tasks $|s\Gamma_i|$ associated with the user task τ_i can vary between 1 and the number of instances that the task executes during its period T_i. For instance, the user task τ_4 (an actor that composes the *h.263* SDFG) can have between 1 and 99 scrubbing tasks associated, depending on the value of Υ. After defining the scrubbing task sets ($s\Gamma_i$), the minimum scrubbing periods of each scrubbing task $s\tau_j \in \{s\Gamma_0 \cup ... \cup s\Gamma_{|\Gamma|-1}\}$ associated with all user tasks implemented in the system are computed. The defined periods have to make the entire scrubbing process schedulable by the EDL scheduling algorithm. In this particular case, the scrubbing periods are found assuming that only 30% of the ICAP port can be

[1]The number of configuration frames used by each user task is estimated according their complexity.

Table II. Scrubbing Task Sets ($s\Gamma_i$) Parameters—$\Upsilon = 11.0$ms

$s\tau_j$	$s\Gamma_i$	$SC_j(ms)$	$ST_j(ms)$	$\Phi_j(ms)$	$\zeta_j(cu)$
$s\tau_0$	$s\Gamma_0$	0.2025	50.000	3.700	0.250
$s\tau_1$	$s\Gamma_1$	0.1215	100.00	1.800	0.250
$s\tau_2$	$s\Gamma_2$	0.0810	100.00	1.900	0.250
$s\tau_3$	$s\Gamma_3$	0.8100	10.345	9.535	0.050
$s\tau_4$	$s\Gamma_4$	0.0340	10.345	1.829	0.050
$s\tau_5$	$s\Gamma_5$	0.0251	10.345	1.929	0.050
$s\tau_6$	$s\Gamma_6$	0.0527	10.345	10.105	0.050
$s\tau_7$	$s\Gamma_7$	0.0211	10.345	10.239	0.050

Table III. Scrubbing Task Sets ($s\Gamma_i$) Parameters—$\Upsilon = 4.13$ms

$s\tau_j$	$s\Gamma_i$	SC_j(ms)	ST_j(ms)	Φ_j(ms)	ζ_j(cu)
$s\tau_0$	$s\Gamma_0$	0.2025	50.000	3.700	0.250
$s\tau_1$	$s\Gamma_1$	0.1215	100.00	1.800	0.250
$s\tau_2$	$s\Gamma_2$	0.0810	100.00	1.900	0.250
$s\tau_3$	$s\Gamma_3$	0.8100	10.345	9.535	0.050
$s\tau_4$	$s\Gamma_4$	0.0340	10.345	1.829	0.050
$s\tau_5$	$s\Gamma_4$	0.0340	10.345	5.945	0.050
$s\tau_6$	$s\Gamma_5$	0.0251	10.345	1.929	0.050
$s\tau_7$	$s\Gamma_5$	0.0251	10.345	6.045	0.050
$s\tau_8$	$s\Gamma_6$	0.0527	10.345	10.105	0.050
$s\tau_9$	$s\Gamma_7$	0.0211	10.345	10.239	0.050

used by the scrubbing process. Tables II and III present the scrubbing tasks' parameters given by the proposed mechanism (in particular the scrubbing tasks $s\tau_j$ associated with each user task τ_i and their scrubbing periods ST_j). Table II presents the scrubbing task set considering a maximum time distance between the user task execution and the last corresponding scrubbing execution (Υ) equal to 11ms. Please note that, in this case, only one scrubbing task is associated with each user task τ_i. On the other hand, Table III presents the scrubbing task set, when $\Upsilon = 4.13$ms. In this case, two scrubbing tasks are associated with the user tasks τ_4 and τ_5. Figure 11 describes the scrubbing execution according to the proposed scrubbing mechanism. Figure 11(II) describes the execution of the scrubbing task set when $\Upsilon = 11$ms. Figure 11(III) presents the scrubbing task set execution when $\Upsilon = 4.13$ms.

After determining the scrubbing task periods, the scrubbing schedule is computed and executed during runtime in windows following the EDL algorithm. According to this scheduling algorithm, the scrubbing tasks are executed as late as possible, nearest to the deadline/period (ST_j). Therefore, the scrubbing tasks execute in the best case just before the corresponding user task execution, minimizing the probability of the user task being affected by an SEU fault. The size of the scrubbing schedule window is set to 1s ($\Delta = 1$s) and the auxiliary window is equal to zero ($\Omega = 0$). The scrubbing schedule window is relatively large, since the objective of the case study is to evaluate the enhanced proposed scrubbing mechanism when applied to heterogeneous applications and not the execution of the scrubbing process in windows. The larger the size of the scrubbing windows, the closer we are to the optimal solution. The impact of the scrubbing window size on the system reliability metric will be assessed in more detail in the experimental results section.

The proposed scrubbing mechanism (*Scheduled*) is evaluated comparing it with the other existing scrubbing mechanisms that can be divided in two classes, *Selective* and *Blind*, as explained in Section 2.2. Please note that with 30% of ICAP capacity available,

Table IV. System Reliability Metric over Υ. Criticality Is Assigned According to the User Criteria (*cu* Method)

	Υ(ms)									
Method	0.2	0.4	0.6	0.8	1.0	1.2	1.4	1.6	1.8	2.0
Scheduled	**0.99**	**0.99**	**0.98**	**0.98**	**0.98**	**0.97**	**0.97**	**0.97**	**0.96**	**0.96**
Selective	0.93	0.93	0.93	0.93	0.93	0.93	0.93	0.93	0.93	0.93
Blind	0.79	0.79	0.79	0.79	0.79	0.79	0.79	0.79	0.79	0.79

each user task of this case study is scrubbed every 4.5ms using the *Selective* approaches and every 115.3ms using the *Blind* ones.

Table IV compares the system reliability metric when the proposed (*Scheduled*), *Selective*, and *Blind* scrubbing mechanisms are applied to this particular case study and when the criticality is assigned according to the user criteria (*cu*). In this case, all the applications have the same importance to the system. The system reliability metric is computed for 24 hours through Equation (4) and considering an Υ ranging from 0.2ms to 2ms. Please observe that the proposed mechanism always performs better when compared to the other solutions. This occurs due to the fact that they execute without any relation to the user applications/tasks. Considering also $\Upsilon = 0.2$ms, the proposed mechanism performs 6% and 20% better than the *Selective* and *Blind* scrubbing mechanisms, respectively. Please note that when the proposed scrubbing mechanism is applied, the system reliability metric decreases when Υ increases, as expected. This occurs because the number of scrubbing tasks assigned to tasks τ_4 and τ_5 decreases when Υ increases.

7. EXPERIMENTAL RESULTS

Several experiments were conducted in order to better evaluate the proposed scrubbing mechanism. The experiments were based on a Virtex-6 LX240T SRAM-based FPGA with 28,464 configuration frames. It was assumed that each frame is scrubbed at the maximum ICAP frequency (100MHz). Therefore, each FPGA configuration frame requires 0.81μs to be scrubbed. Moreover, for these experiments the FPGA device was simulated to be placed in the space environment, subjected to SEUs with a rate $\lambda = \frac{1}{1Hour}$ [Bridgford et al. 2008].

For the experiments (expected the last one), the implementation of an application that includes one SDFG G (composed with up to 5 actors) and 5 independent periodic applications was simulated. Therefore, the number of user tasks implemented in the FPGA can reach up 10 user tasks. The SDFG is synthetically generated, using the function *sdf3generate-sdf* available in the SDF3 framework [2015]. The SDFG is generated with the following parameters: $stronglyConnected =$ "$true$," $acyclic =$ "$false$," and $multigraph =$ "$false$." Moreover, the total number of actor execution instances during the iteration period ($\sum_{i \in V_G} |EP_i|$) can be up to 25. The actors that compose this graph G are also synthetically generated. The execution time of each actor assumes values that are multiples of 0.1ms and are uniformly distributed between 0.5ms and 1.5ms. The number of configuration frames used to implement each user task, either actor or independent period task, assumes values that are multiples of 10. They are also uniformly distributed between 10 and 500, corresponding to a scrubbing execution time (SC) between 8.1μs and 405μs, respectively. Moreover, the periods of each generated periodic task assumes only values that are multiples of 1ms and uniformly distributed between 10ms and 50ms.

For each experiment (except the last one), 1,000 user applications were synthetically generated, according to the features and parameters described above. The proposed scrubbing mechanism (*Scheduled*) is also compared to the existing solutions, classified as *Selective* and *Blind*. All the mechanisms are evaluated using the *cu* method to assign

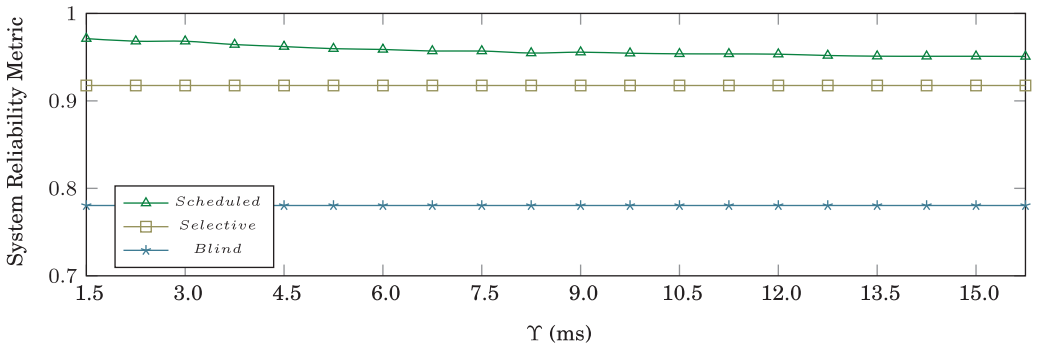

Fig. 12. System reliability metric over ϒ. Criticality is assigned according to the user criteria (*cu*). All the user applications have the same criticality in this experiment.

the user task's criticality. In the first four experiments, it is assumed that the *cu* method assigns the same criticality/importance to all the user applications.

7.1. System Reliability Metric Over ϒ

This first experiment evaluates the system reliability metric over the maximum time distance (ϒ) recommended between one user task execution instance and the corresponding latest scrubbing execution. Please note that the previous scrubbing mechanism presented in Santos et al. [2014] has proposed the scrubbing schedule computation for the LCM interval given by the periods of all scrubbing tasks. Despite the fact that this solution is optimal, when it is applied to heterogeneous applications with different execution behaviors and very different periods, the LCM given by the corresponding scrubbing tasks can be huge. Therefore, computing the scrubbing schedule for huge LCM intervals can be computationally very expensive and impractical. This has occurred in most of the user applications synthetically generated. Therefore, the scrubbing schedule window size (Δ) for this experiment is set to a considerably large value, 1 minute, in order to get results as close as possible to the optimal ones. The auxiliary window (Ω) is set to 0. Please note that all tested scrubbing mechanisms can only use 30% of the overall ICAP utilization available.

For each user application generated, the system reliability metric is computed for 24 hours, considering ϒ ranging from 1.5ms to 15.75ms. Figure 12 shows the obtained results. As expected, for bigger ϒ*s* time distances the system reliability metric is smaller, since the proposed scrubbing system generates fewer scrubbing tasks associated with each user task that composes the SDFG *G*. In the best case (with ϒ = 1.5ms), the proposed scrubbing mechanism (*Scheduled*) performs 6.0% and 19.1% better when compared to *Selective* and *Blind* approaches, respectively.

7.2. System Reliability Metric Over the ICAP Utilization

This experiment evaluates the system reliability metric over the ICAP utilization provided to the scrubbing process. For each generated user application, the system reliability metric was computed for ICAP utilization ranging from 20% and 100%. Similarly to the previous experiment, the scrubbing schedule window size (Δ) is set to 1 *minute* and the auxiliary window (Ω) equal to 0.

Figure 13 shows the respective results measured for 24 hours. As expected, the system reliability metric decreases when the ICAP utilization is constrained. Moreover, it is important to note that the proposed solution always performs better than the other mechanisms. In particular, the proposed solution performs on average 3.8% and 16.3%

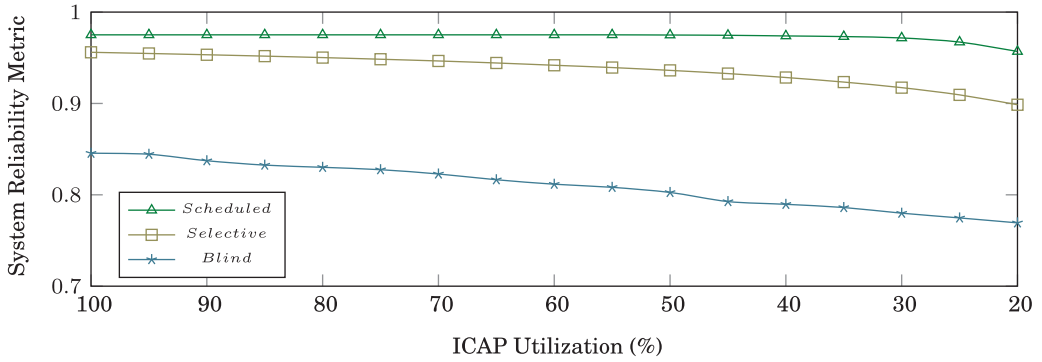

Fig. 13. System reliability metric over the ICAP utilization. Criticality is assigned according to the user criteria (*cu*). All the user applications have the same criticality in this experiment.

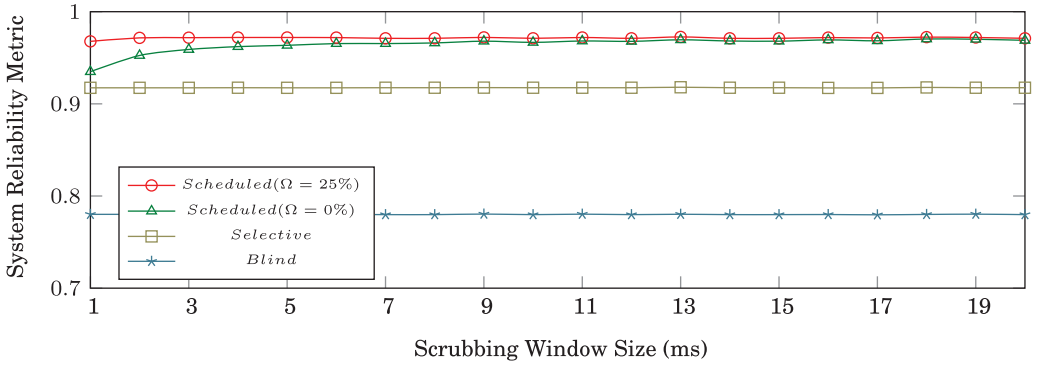

Fig. 14. System reliability metric over the scrubbing schedule window size. Criticality is assigned according to the user criteria (*cu*). All the user applications have the same criticality in this experiment.

better (with a maximum of 5.8% and 19.2%) when compared to the *Selective* and *Blind* approaches, respectively.

7.3. System Reliability Metric Over the Schedule Window Size (Δ)

Using the same user applications considered in the previous experiments, this experiment evaluates the system reliability metric over the scrubbing schedule window size (Δ). Please note that for this experiment the maximum ICAP utilization provided to the scrubbing process is also 30%. Moreover, ϒ is set to 1.5ms, the best case measured in the first experiment.

Figure 14 shows the system reliability metric results measured for 24 hours and considering a schedule window size (Δ) ranging from 1ms to 20ms. As expected, the system reliability metric using the proposed approach increases with the increase of the scrubbing schedule window size. The proposed solution is also evaluated using the auxiliary window mechanism (Ω) equal to 25% of the main scrubbing schedule window. Applying the auxiliary window to the proposed scrubbing mechanism improves the system reliability metric up to 3.5%. Moreover, it allows us to achieve for bigger Δs (close to 20ms) the best system reliability metric measured on the first experiment with a ($\Delta =$ 1 minute and $\Upsilon = 1.5$ms). Therefore, $\Delta <= 20$ms leads to a memory overhead reduction to store the scrubbing schedule (as shown in the next experiment) without a significant impact on the system reliability metric and it allows us to quickly update the scrubbing schedule according to any modification that may occur on the user application. When

Table V. Scrubbing Schedule Computation Time over Υ

Υ(ms)	1.5	3.0	4.5	6.0	7.5	9.0	10.5	12.0	13.5	15.0
***CPU Time* / Δ**	**0.05**	**0.05**	**0.04**	**0.04**	**0.03**	**0.03**	**0.03**	**0.03**	**0.03**	**0.03**
N. of scrubbing tasks	**17.5**	**14.9**	**12.4**	**11.4**	**10.9**	**10.5**	**10.2**	**10.0**	**9.5**	**9.4**

Table VI. Scrubbing Schedule Computation Time over the Window Size (Δ)

Window Size Δ(ms)	1	3	5	7	9	11	13	15	17	19
***CPU Time* / Δ**	**0.30**	**0.13**	**0.11**	**0.08**	**0.07**	**0.07**	**0.06**	**0.06**	**0.05**	**0.05**

the proposed scrubbing mechanism (with auxiliary window) is compared to the other current scrubbing approaches *Selective* and *Blind*, improvements up to 5.9% and 18.8% are observed.

7.4. Scrubbing Scheduler Evaluation

This set of experiments evaluates the scrubbing scheduler and demonstrates its feasibility. The following experiments were executed using a machine consisting of a Intel(R) Xeon(R) processor running at a constant speed of 2.40GHz and 8GB of memory.

7.4.1. Memory Overhead. This experiment evaluates the proposed scrubbing mechanism in terms of memory overhead required to store the produced scrubbing schedule. In this sense, this experiment compares the memory required to store the scrubbing schedule when Δ ranges between 1ms and 20ms to the memory required to store the scrubbing schedule when Δ = 1minute (window size considered in the first experiment). It is observed that for a scrubbing schedule window size equal to 20ms (the biggest considered), the memory overhead required $163B$ using the *cu* method to assign the criticality. Moreover, considering the previous experiment (7.3), for window sizes bigger than 5ms, the difference to the optimal solutions (with Δ = 1minute) is less than 1%, reaching almost 0% for window sizes equal to 20ms. Therefore, it is concluded that by using smaller scrubbing window sizes, there is a significant reduction in the memory overhead required to store the scrubbing schedule (for Δ = 1minute, the memory used is above 500KB) with insignificant losses in terms of system reliability metric.

7.4.2. Scrubbing Schedule Computation Time Over Υ. This experiment evaluates the behavior of the scrubbing schedule computation time over the variation of Υ. Table V presents the results, taking into account the same experiment presented in 7.1. Please note that in this experiment Δ is equal to 20ms. As it is possible to observe, the time to compute the scrubbing schedule decreases when Υ increases. When Υ is smaller, the proposed scrubbing mechanism generates a higher number of scrubbing tasks. Therefore, the time to compute the respective scrubbing schedule for a Δ window size is higher when compared to smaller Υs.

7.4.3. Scrubbing Schedule Computation Time over the Window Size. This experiment evaluates the behavior of the time to compute the scrubbing schedule over the window size (Δ). Based on the experiment described in Section 7.3, Table VI presents the ratio between the time required to compute the scrubbing schedule and the the window size. As you can observe the ratio reduces significantly with the increase of the window size. Please note that for Δ = 20ms the scrubbing schedule can be computed using 5% of the window size.

7.5. Dynamic Adaptation

This experiment assesses the impact of user tasks modifications on their reliability. For that, five periodic independent tasks were synthetically generated, and their

Table VII. Reliabiliy of the user Task τ_i Over Several user Task Set Modifications

	Scheduled				*Selective*				*Blind*			
Task	A	B	C	D	A	B	C	D	A	B	C	D
τ_1	0.99	**0.86**	**0.90**	**0.99**	0.93	0.93	0.93	0.93	0.82	0.82	0.82	0.82
τ_2	0.84	0.84	**0.99**	**0.99**	0.78	0.78	0.78	0.78	0.71	0.71	0.71	0.71
τ_3	0.99	0.99	**0.99**	**0.99**	0.97	0.97	0.97	0.97	0.96	0.96	0.96	0.96
τ_4	0.99	0.99	**0.99**	—	0.95	0.95	0.95	—	0.90	0.90	0.90	—
τ_5	0.91	**0.99**	—	—	0.87	0.87	—	—	0.84	0.84	—	—

implementation in FPGA was simulated. The criticality of the task, assigned by the user, is descending according to their ids. Therefore, the task τ_1 is the most critical and task τ_5 is the least critical one. The ICAP utilization provided to the scrubbing process was set to 5% and Δ was configured to 20ms. Table VII shows the reliability of each task when the different scrubbing mechanisms are applied. Column A shows the reliability of each task when the system starts the execution. At a certain instant during the execution, for some reason the criticality of task τ_1 is switched with τ_5. Task τ_5 becomes the most important in the system and task τ_1 the least important one. As you can observe in column B, when the proposed scrubbing mechanism is applied, the reliability of task τ_5 increases automatically, since more scrubbing effort is given to this task as opposed to task τ_1. During also the execution, task τ_5 and task τ_4 are suspended sequentially (columns C and D, respectively). The reduction of the task set allows the proposed mechanism to automatically redistribute the scrubbing effort given to task τ_4 and τ_5 by the remaining active user tasks, increasing their reliability for the same ICAP utilization. Considering the other existing scrubbing mechanisms, the reliability of the user tasks remains unchanged over the several modifications, since these mechanism are completely static and decoupled from the user tasks' execution.

8. CONCLUSIONS

In this article, a new scrubbing mechanism is proposed in order to improve the system reliability metric of heterogeneous embedded system applications, composed of periodic/sporadic and streaming/DSP user applications, implemented in reconfigurable devices. This new mechanism takes into account each user application execution behavior as well as its criticality in the system to find the proper scrubbing schedule that maximizes the overall system reliability metric. The scrubbing schedule is computed and executed in windows of configurable size. This mechanism allows the reduction of the memory overhead required to store the scrubbing schedule, as well as it allows dynamic adaptations on the scrubbing process as response to user task reconfigurations/modifications at runtime.

Conducted experiments with different functional scenarios show the feasibility of the proposed scrubbing mechanism. They show important improvements on the system reliability metric compared to other scrubbing mechanisms. Improvements up to 19.2% on the system reliability metric are observed on average. The experiments also show significant reductions on the memory overhead with small impact on the system reliability metric when the scrubbing schedule is performed in windows. This mechanism also allows high reactivity on the scrubbing process when any modification occurs on the user application.

As future work, the computation of the ratio between the system reliability metric achieved by the different scrubbing approaches and their energy consumption would also be a useful metric for comparison.

REFERENCES

C. Argyrides, D. K. Pradhan, and T. Kocak. 2011. Matrix codes for reliable and cost efficient memory chips. *IEEE Trans. VLSI Syst.* 19, 3, 420–428.

P. Axer, M. Sebastian, and R. Ernst. 2011. 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)*.

S. Baruah, Haohan Li, and L. Stougie. 2010. Towards the design of certifiable mixed-criticality systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'10)*.

C. Bolchini, M. Miele, and M. Santambrogio. 2007. TMR and partial dynamic reconfiguration to mitigate SEU faults in FPGAs. In *Defect and Fault-Tolerance in VLSI Systems (DFT'07)*.

C. Bolchini and C. Sandionigi. 2014. Design of hardened embedded systems on Multi-FPGA platforms. *ACM Trans. De. Autom. Electron. Syst.* 20, 1, 16:1–16:26.

B. Bridgford, C. Carmichael, and C. Tseng. 2008. *Single-Event Upset Mitigation Selection Guide*.

F. Brosser, E. Milh, V. Geijer, and P. Larsson-Edefors. 2014. Assessing scrubbing techniques for Xilinx SRAM-based FPGAs in space applications. In *IEEE International Conference on Field-Programmable Technology (FPT'14)*.

A. Burns and A. Wellings. 1995. *HRT-HOOD^{TM}: A Structured Design Method for Hard Real-Time Ada Systems*, Volume 3 1st Edition. 312.

E. Cannella, M. Bamakhrama, and T. Stefanov. 2014. System-level scheduling of real-time streaming applications using a semi-partitioned approach. In *Design, Automation & Test in Europe (DATE'14)*.

C. Carmichael, M. Caffrey, and A. Salazar. 2000. *Correcting Single-Event Upsets Through Virtex Partial Configuration*. Technical Report. Xilinx.

J. Cheatham, J. Emmert, and S. Baumgart. 2006. A survey of fault tolerant methodologies for FPGAs. *ACM Trans. Des. Autom. Electron. Syst.* 11, 2, 501–533.

H. Chetto and M. Chetto. 1989. Some results of the earliest deadline scheduling algorithm. *IEEE Trans. Softw. Eng.* 15, 10, 1261–1269.

A. Das, A. Kumar, and B. Veeravalli. 2013. 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)*.

R. Dobrin, H. Aysan, and S. Punnekkat. 2008. Maximizing the fault tolerance capability of fixed priority schedules. In *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'08)*.

J. Forget, F. Boniol, E. Grolleau, D. Lesens, and C. Pagetti. 2010. Scheduling dependent periodic tasks without synchronization mechanisms. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'10)*.

A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, A. J. M. Moonen, M. J. G. Bekooij, B. D. Theelen, and M. R. Mousavi. 2006. Throughput analysis of synchronous data flow graphs. In *International Conference on Application of Concurrency to System Design (ACSD'06)*.

N. Guan, Q. Deng, Z. Gu, W. Xu, and G. Yu. 2008. Schedulability analysis of preemptive and nonpreemptive EDF on partial runtime-reconfigurable FPGAs. *ACM Trans. Des. Autom. Electron. Syst.* 13, 4, 56:1–56:43.

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

P. Hölzenspies, J. Hurink, J. Kuper, and G. Smit. 2008. Run-time spatial mapping of streaming applications to a heterogeneous multi-processor system-on-chip (MPSoC). In *Conference on Design, Automation and Test in Europe (DATE'08)*.

K. A. Hoque, O. A. Mohamed, Y. Savaria, and C. Thibeault. 2014. Probabilistic model checking based DAL analysis to optimize a combined TMR-blind-scrubbing mitigation technique for FPGA-based aerospace applications. In *ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'14)*.

K. Jeffay, D. F. Stanat, and C.U. Martel. 1991. On non-preemptive scheduling of period and sporadic tasks. In *IEEE International Real-Time Systems Symposium (RTSS'91)*.

N. Jing, J. Lee, Z. Feng, W. He, Z. Mao, and L. He. 2013. SEU fault evaluation and characteristics for sram-based FPGA architectures and synthesis algorithms. *ACM Trans. Des. Autom. Electron. Syst.* 18, 1, 13:1–13:18.

J. Johnson, W. Howes, M. Wirthlin, D. McMurtrey, M. Caffrey, P. Graham, and K. Morgan. 2008. Using duplication with compare for on-line error detection in FPGA-based designs. In *IEEE International Aerospace Conference (AeroConf'08)*.

K. M. Kavi, B. P. Buckles, and U. N. Bhat. 1986. A formal definition of data flow graph models. *IEEE Trans. Comput.* C-35, 11, 940–948.

I. Koren and C. M. Krishna. 2007. *Fault-Tolerant Systems*. Morgan Kaufmann.

Akash Kumar, Shakith Fernando, Yajun Ha, Bart Mesman, and Henk Corporaal. 2008. Multiprocessor systems synthesis for multiple use-cases of multiple applications on FPGA. *ACM Trans. Des. Autom. Electron. Syst.* 13, 3, 40:1–40:27.

M. Lanuzza, P. Zicari, F. Frustaci, S. Perri, and P. Corsonello. 2010. 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)*.

Lattice Semiconductor. 2015. Homepage. Retrieved from http://www.latticesemi.com/products/fpga/index.cfm.

J. Leung and M. Merrill. 1980. A note on preemptive scheduling of periodic, real-time tasks. *Inform. Process. Lett.* (1980).

D. Liu, S. Yin, L. Liu, and S. Wei. 2013. Polyhedral model based mapping optimization of loop nests for CGRAs. In *Annual Design Automation Conference (DAC'13)*.

D. Liu, S. Yin, Y. Peng, L. Liu, and S. Wei. 2015. Optimizing spatial mapping of nested loop for coarse-grained reconfigurable architectures. *IEEE Trans. VLSI Syst.* 23, 11, 2581–2594.

D. McMurtrey, K. Morgan, B. Pratt, and M. Wirthlin. 2008. *Estimating TMR Reliabililty on FPGAs Using Markov Models*. Technical Report.

G. L. Nazar, L. P. Santos, and L. Carro. 2013a. Accelerated FPGA repair through shifted scrubbing. In *IEEE International Conference on Field Programmable Logic and Applications (FPL'13)*.

G. L. Nazar, L. P. Santos, and L. Carro. 2013b. Scrubbing unit repositioning for fast error repair in FPGAs. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'13)*.

G. Neuberger, F. de Lima, L. Carro, and R. Reis. 2003. A multiple bit upset tolerant SRAM memory. *ACM Trans. Des. Autom. Electron. Syst.* 8, 4, 577–590.

S. Park, D. Lee, and K. Roy. 2012. Soft-error-resilient FPGAs using built-in 2-D hamming product code. *IEEE Trans. VLSI Syst.* 20, 2, 248–256.

B. Pratt, M. Caffrey, P. Graham, K. Morgan, and M. Wirthlin. 2006. Improving FPGA design robustness with partial TMR. In *IEEE International Reliability Physics Symposium (IRPS'06)*.

R. Santos, S. Venkataraman, A. Das, and A. Kumar. 2014. Criticality-aware scrubbing mechanism for SRAM-based FPGAs. In *IEEE International Conference on Field Programmable Logic and Applications (FPL'14)*.

R. Santos, S. Venkataraman, and A. Kumar. 2015. Dynamically adaptive scrubbing mechanism for improved reliability in reconfigurable embedded systems. In *Design Automation Conference (DAC'15)*.

A. Sari and M. Psarakis. 2011. Scrubbing-based SEU mitigation approach for systems-on-programmable-chips. In *IEEE International Conference on Field-Programmable Technology (FPT'11)*.

A. Sari, M. Psarakis, and D. Gizopoulos. 2013. Combining checkpointing and scrubbing in FPGA-based real-time systems. In *IEEE VLSI Test Symposium (VTS'13)*.

A. Sarkar, F. Mueller, H. Ramaprasad, and S. Mohan. 2009. Push-assisted migration of real-time tasks in multi-core processors. *SIGPLAN Not.* 44, 7, 80–89.

SDF3 Framework. 2015. Homepage. Retrieved from http://http://www.es.ele.tue.nl/sdf3/.

F. Siegle, T. Vladimirova, J. Ilstad, and O. Emam. 2015. Mitigation of radiation effects in SRAM-based FPGAs for space applications. *ACM Computing Surveys (CSUR)* 47, 2, 37:1–37:34.

S. Venkataraman, R. Santos, S. Maheshwari, and A. Kumar. 2014. Multi-directional error correction schemes for sram-based FPGAs. In *IEEE International Conference on Field Programmable Logic and Applications (FPL'14)*.

Xilinx Corporation. 2012. *Partial Reconfiguration User Guide UG702 (v14.1)*.

Xilinx Corporation. 2015. *Virtex-6 FPGA Configuration—User Guide UG360 (v3.9)*.

Received February 2016; revised September 2016; accepted September 2016