Reliability Improvement of Hardware Task Graphs via Configuration Early-fetch

Reza Ramezani, Yasser Sedaghat, and Juan Antonio Clemente

***Abstract*—This study presents a technique to improve the reliability and the Mean Time to Failure (MTTF) of hardware task graphs running on reconfigurable computers. This technique, which has been named *Task Early-fetch*, can be applied to a sequence of one or several applications, represented as task graphs. It consists in carrying out the reconfiguration of some tasks within the execution of the previous task graph, plus increasing the redundancy level of the early-fetched tasks. Experimental results on actual task graphs show the positive impacts of the proposed technique. Thus, without deteriorating the execution time (makespan), on average, a 114% MTTF improvement is achieved for no-fault-tolerant task graphs, and the improvement is more significant when applying to fault-tolerant task graphs. Finally, this paper presents a hardware implementation of a manager that applies these techniques at run-time and steers the execution of the running task graphs. It demonstrates that, with 0.03% consumption of FFs and LUTs and also 1.22% occupancy of BRAMs available on a Xilinx Virtex UltraScale XCVU095-2FFVA2104E FPGA, the required run-time computations can be carried out in negligible delays.**

***Index Terms*—FPGAs, Reliability, Task Graph, Scheduling, Early-fetch, Fault Tolerance.**

I. INTRODUCTION

SRAM-based Field Programmable Gate Arrays (FPGAs) have recently drawn the attention of researchers and manufacturers of complex electronic systems in fields such as avionics and aerospace [1]. The reason is that, unlike Application-Specific Integrated Circuits (ASICs), FPGAs can be reconfigured multiple times during the mission and also feature lower cost than ASICs, as well as less time to market [2]. Partial reconfigurability makes FPGAs able to configure only a portion of the device while the remaining resources continue their normal operation. In order to execute multiple functionalities in a time-multiplexed manner, a scheduler is required to steer the execution of the hardware tasks [3].

Partially Run-time Reconfigurable (PRR) FPGAs suffer from reconfiguration delay and also susceptibility to the consequences of the Single Event Effects (SEEs) [4]. To alleviate the susceptibility to consequences of SEE, Fault Tolerance (FT) techniques are required to increase the reliability of a given design, but in most of the cases, they also come at the cost of degrading the system's performance. Therefore, reliability and performance should be optimized simultaneously.

This paper aims at improving the reliability of applications, represented as task graphs, running on FPGA-based reconfigurable computers, without deteriorating their execution time –which is known as *Makespan*–. For this purpose, a novel technique, named *Task Early-fetch* is presented. It consists in carrying out two modifications on a pair of consecutive task-graph schedules. On the one hand, it loads the configuration data of some tasks of a given task graph within the execution of the previous one. On the other hand, it increases the redundancy level of the involved tasks to improve their reliability, without deteriorating the makespan. In this work, it is assumed a dynamic environment with one or more known task graphs at design time, but an unknown execution order at run time.

The experiments on actual task graphs show that the proposed technique improves the reliability and Mean Time to Failure (MTTF) of the task graphs without deteriorating their makespan. Additional experiments on hardware tasks in fault-varying environments show that the proposed technique outperforms other state-of-the-art FT techniques [5], [6]. Finally, the hardware implementation that has been presented demonstrates that, with a very affordable hardware cost, the run-time computation required to implement the proposed technique is negligible.

The remainder of this paper is organized as follows. Section II introduces some related work, and Section III shows illustrative examples. Next, Section IV describes the proposed *Early-fetch* technique. Experimental results are shown in Section V and finally, the paper concludes in Section VI.

II. RELATED WORK

Many researchers have investigated the FT issues in FPGAs, which can be categorized into three groups of mitigation approaches, namely: design-based methods, placement- and routing-based methods, and recovery-based ones.

Design-based methods are typically built upon redundancy, which is a very effective approach to mitigate soft errors [7], especially in environments with dynamic fault rates [5]. These methods use different replications, at different granularities, to increase the system reliability [8]. In this regard, different fine and coarse grain redundancy-based FT techniques for space-computing systems have been investigated in [9], [10]. As an example of the application of FT techniques in FPGAs, a system-level Duplication With Compare (DWC) FT technique has been presented in [11] to improve the reliability of adaptive

Manuscript received June 20, 2016; revised 24 August, 2016; accepted October 29, 2016.

This work was supported in part by the Ministry of Science, Research and Technology of Iran and by the Spanish MCINN project TIN2013-40968-P, *Corresponding author: Yasser Sedaghat*

R. Ramezani and Y. Sedaghat are with the Department of Computer Engineering, Ferdowsi University of Mashhad, Mashhad, Iran (e-mails: reza.ramezani@mail.um.ac.ir, y_sedaghat@um.ac.ir).

J. A. Clemente is with the Computer Architecture Department, Universidad Complutense de Madrid, Madrid 28040, Spain (e-mail: ja.clemente@fdi.ucm.es).

equalizers, implemented on FPGAs. In a similar approach, [12] presents a redundant FPGA-implemented speed controller core for high speed trains. The application of both TMR and DWC approaches combined with a check-pointing technique to build reliable soft processors has also been investigated in [13].

Placement- and routing-based FT techniques increase the reliability of a design by adapting traditional place&route techniques for FPGAs for harsh environments, at different design phases. For example, an interesting technique has been presented in [14] which manages the signals between functions in such a way that multiple errors affecting two different connections are not possible. In a similar approach, [15] studies both fault occurrence and error propagation probabilities to propose a reliability-oriented placement and routing algorithm. Anyway, all these techniques can be applied to a given hardware task and, as indicated by [16], they can be used in combination with other design-based methods to increase the reliability of the circuits.

However, the aforementioned techniques cannot prevent fault accumulation at run-time. Recovery-based methods are designed to resolve the fault accumulation problem [17]. Most of these techniques are based on recovering the value of the faulty cells [18]. For example, the studies in [19], [20] determine different scrubbing rates for different circuits, based on their failure rate, in such a way that the system reliability is maximized. Some other techniques are based upon replacing the faulty blocks with the previously generated ones, which are functionally equivalent block instances, that do not use the faulty resources [21], [22].

Combining design-based and recovery-based methods is very effective for mitigating soft errors in FPGA-based systems. For example, [23] addresses the problem of tolerating *N* failures in nano-satellite swarm-based systems, using spare swarms. This work presents general ideas that are not particularly focused on a specific device, but they can be applied to reconfigurable computers. Similarly, [24] employs a redundancy-based approach to employ spare units in which each task has many redundancies, so that some of them are active and in order to reduce power consumption the remaining are standby. The work by Yousuf [25] is another study in this area that combines hardware and software tasks to guarantee a given target reliability, while reducing the energy consumption. A similar study, has been done by [26], [27] which introduces a task partitioning scheme to tolerate transient and permanent faults for software/hardware tasks in heterogeneous and reconfigurable platforms.

In embedded systems in general, and in FPGAs in particular, applications are usually represented as a Directed Acyclic Graph (DAG) or a Task Graph (TG), whose nodes represent computational tasks and whose edges represent dependencies among tasks. When such task graphs run on reconfigurable computers, they have to be scheduled in a way that both tasks precedence constraints and the resource limitations are met. This requires to take task scheduling and task placement into account [28], [29]. The performance of the scheduling methods could be improved by employing *Task Prefetch* [30], or *Task Reuse* techniques [31]. These techniques configure a given task in advance [32], and they can be used to improve the makespan of task graphs [33], as well as alleviating the fragmentation problem of FPGAs [34]. However, as it will be discussed, these techniques have adverse effects on the task reliability. By prefetching a task, its residency time increases on FPGA, which as a result, increases the time that the task is exposed to radiations. In these works, the negative effects of the prefetch technique on the task reliability has not been evaluated nor taken into account.

TABLE I
CHARACTERISTICS OF THE TASK GRAPH DEPICTED IN FIGURE 1. THREE FT STRATEGIES ARE APPLIED: SINGLE MODULE (NO REDUNDANCY), DWC (DUPLICATION WITH COMPARE) AND TMR (TRIPLE MODULAR REDUNDANCY)

Task	**Computation Time (*ms*)**	**Resources Occupancy (%)**	**Configuration Delay (*ms*)**	**FT Strategy**
τ_1	140	20	80	Single
τ_2	439	14	56	DWC
τ_3	147	10	40	Single
τ_4	596	16	64	TMR
τ_5	300	14	56	DWC

The scheduling methods can also been enhanced to take FT requirements into account. These techniques aim at guaranteeing a given system performance whereas the system reliability is increased as well. For example, a *Primary/Backup* scheme is proposed in [35] in which two versions of a task run with minimum time overlap. In [36], a real-time fault-tolerant scheduling algorithm is proposed which, schedules hybrid tasks able to tolerate f_i faults during task execution. Our previous study [6] showed that, by using optimization methods and choosing, for each task, the proper FT technique from the Pareto-set (referred to as *Pareto-based FT techniques*), it is possible to increase the reliability of task graphs without deteriorating their makespan. The application of different FT strategies on different real-time scheduling algorithms in reconfigurable computers have been investigated in [37]. In order to manage these issues at run-time, some operating systems have been introduced in [38], [39] which provide an environment for the execution of hardware tasks by considering task communication, task placement, and especially task fault tolerance [40].

This work presents a novel technique, named *Task Early-fetch*, which aims at increasing the reliability of applications, represented as task graphs scheduled on a FPGA-based reconfigurable computer, without deteriorating their makespan.

III. MOTIVATIONAL EXAMPLES

In order to better clarify the proposed technique, this section presents a couple of illustrative examples. In this work, applications are modeled as DAGs and a non-preemptible As Soon As Possible (ASAP) scheduling strategy is used to manage configurations and executions of tasks. It is assumed there exists a set of one or more task graphs in the system and they are executed serially. In this work, the concept of *Stage* refers to a complete execution of a task graph.

The first example assumes the task graph with 5 tasks, depicted in Figure 1, which is executed periodically. Its characteristics have been detailed in Table I.

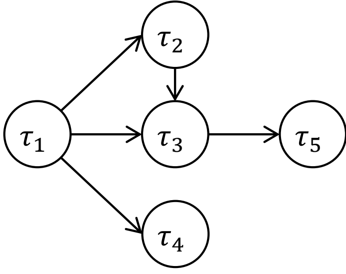

Fig. 1. A sample task graph with 5 tasks

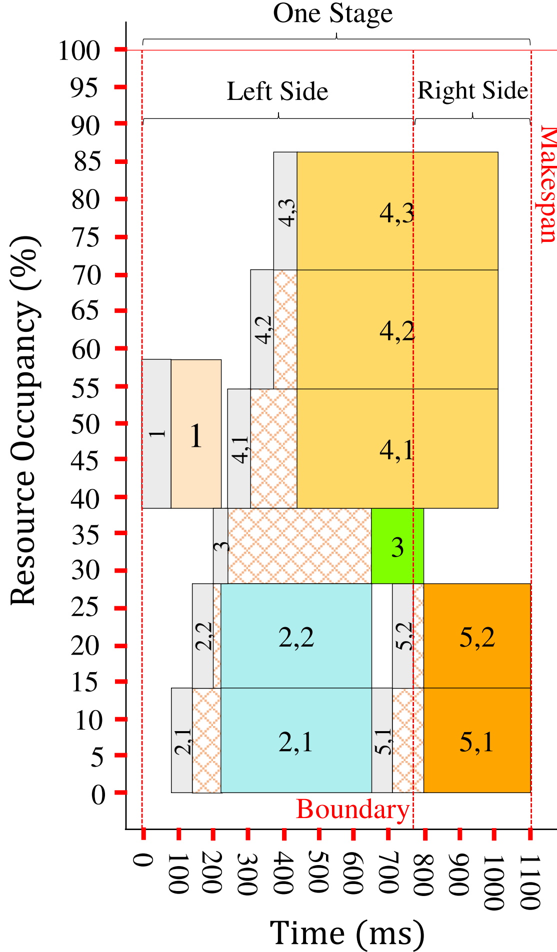

Fig. 2. Example of task graph execution depicted in Figure 1

Figure 2 depicts a simple schedule of the task graph of Figure 1. In this case, the time and area occupied in the reconfigurable computer are presented in the horizontal and vertical axes, respectively. In this figure, the gray-color boxes denote the task configuration delay, whereas the dotted ones indicate that the task has finished its configuration, but it is waiting for its execution to start. Therefore, in this paper it is assumed that the configuration and execution of a given task do not overlap, and a task can start its execution only when it is configured completely.

For the sake of simplicity, in this paper it is also assumed that, at any point, the total area occupied in the reconfigurable computer is simply the addition of the resource consumption of all the tasks that are simultaneously under execution or being reconfigured. This actually depends on many factors, such as the partial reconfiguration model and granularity of the target device, or whether the hardware multitasking system that runs the tasks but implements some sort of task defragmentation. In any case, the technique presented in this paper is orthogonal to all these issues, and one of the many systems that have been proposed in the literature for managing the task graph execution in reconfigurable computers can be used to run the tasks [29], [38], [39], in combination with the presented approach.

As Figure 2 shows, task prefetch allows hiding the reconfiguration delay of some tasks by overlapping them with the execution of other tasks. In addition, an active redundancy-based FT strategy has been applied to tasks τ_2, τ_4 and τ_5. In a prefetch-aware scheduling algorithm, there is a time point in which all the tasks have been configured completely, but the execution of the task graph is not finished yet. In this paper, we have referred to this point as *LastConfigTime* and, in the example of Figure 2, this value is 771 *ms* (i.e., the end of the reconfiguration of $\tau_{5,2}$). In order to define time margins within the schedule to apply the proposed *Early-fetch* technique, a *Boundary* value is defined so that: $LastConfigTime \leq Boundary < Makespan$. The time margin between boundary and makespan can be used to configure some tasks of the next task graph. The criteria of choosing an appropriate value for *Boundary* will be discussed in the next section. Now, let the *Left Side* (LS) and the *Right Side* (RS) of the schedule of task graph TG_i be defined as follows:

- $LS\,(TG_i)$ is the sequence of scheduling orders (i.e., *starting of reconfiguration* and *starting of execution*) comprised between $t = 0$ until $t = Boundary\,(TG_i)$.
- $RS\,(TG_i)$ is the sequence of scheduling orders comprised between $t = Boundary\,(TG_i)$ and $t = Makespan\,(TG_i)$.

Therefore, if there are enough available resources in the target FPGA, the time elapsed within $RS\,(TG_i)$ is a good time margin to carry out the reconfigurations of the early-fetched tasks belonging to the task graph running immediately after TG_i, because no task of TG_i is configured within this time margin. In this example, let us assume $Boundary\,(TG_i) = LastConfigTime\,(TG_i)$.

In order to illustrate the *Early-fetch* technique, Figure 3 shows two successive executions of the task graph presented in Figure 1. In this case, the configuration of one replica of Task τ_2 ($\tau_{2,1}$) of Stage 2 has been early-fetched. In other words, its reconfiguration now takes place within right side of the first stage of the execution. In addition, the configuration delay hidden by this early-fetch has been used to configure another replica of Task τ_2 ($\tau_{2,3}$) at Stage 2. Thus, this technique does not increase the total makespan of the task graph execution in that stage (x-axis). In addition, it does not violate the FPGA size limitation either (y-axis). In this example, further stages of this task graph execution are identical to the second stage of the task graph execution in Figure 3. Finally, note that the right side of both stages is identical, although the redundancy

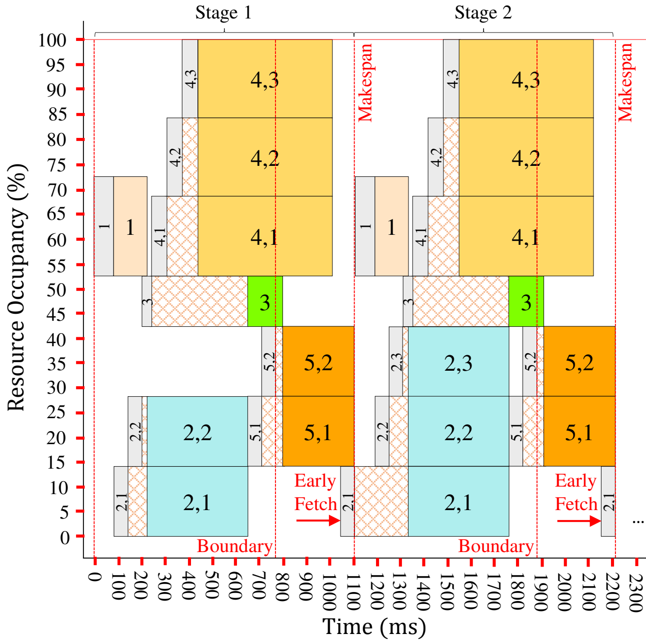

Fig. 3. A two-stage execution of the task graph of Figure 1 with early-fetch and replication of τ_2

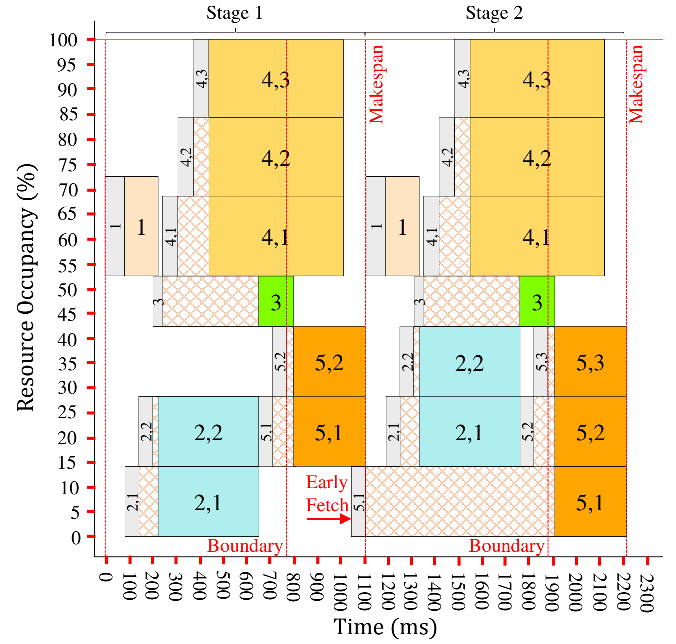

Fig. 4. A two-stage execution of the task graph of Figure 1 with early-fetch and replication of τ_5

level applied to Task τ_2 is different for each stage. The reason is that the early-fetched task (τ_2) is completely executed within $LS\left(TG_i\right)$. Let us bear in mind this fact for the next example.

Figure 4 shows another example in which Task τ_5, whose execution time falls within the right side of the schedule, is early-fetched instead. In this case, the configuration of $\tau_{5,1}$ is early-fetched within Stage 1, and a new replica of that task ($\tau_{5,3}$) is configured at Stage 2. However, as this figure shows, as a consequence of this, there do not exist sufficient resources at the right side of Stage 2 to early-fetch Task $\tau_{5,1}$ from an additional execution of the same task graph (in Stage 3, which is not shown in the figure for simplicity). Therefore, Stage 3 cannot benefit from this technique and its execution would be identical to that of Stage 1. In fact, the reason of this has been the modification of the right side of the schedule at Stage 2, due to the addition of another instance of Task τ_5. In particular, if the following condition is true:

$$\forall \tau_i \in Early_Fetched_Tasks\left(TG\right);$$
$$FinishTime\left(\tau_i\right) \leq Boundary\left(TG\right) \quad (1)$$

then the applicability of the *Early-fetch* technique in Stage i is uniquely dependent on the task graph that runs at Stage $i-1$. However, if this does not happen, i.e.:

$$\exists \tau_i \in Early_Fetched_Tasks\left(TG\right);$$
$$FinishTime\left(\tau_i\right) > Boundary\left(TG\right) \quad (2)$$

then the applicability of the *Early-fetch* technique in Stage i is dependent to the task-graph execution sequence in Stages $[1 \ldots i-1]$. The next section of the paper will explain in detail the consequences of this important fact. It is also noteworthy to remember that the task graphs running in different stages can be the same, or completely different. At any rate, there exists a set of task graphs that can run in the system (which is known in advance), but their execution order is completely unknown at run-time. This assumption is consistent with modern FPGA-based systems, which are dynamically adaptable depending on the run-time requirements [3].

Finally, it is very important to mention that all the modifications introduced in the original schedules are carried out at design time, and no modifications on such schedules are carried out at run-time. As a consequence, the presented approach always works with static schedules. The reason is that, if dynamic schedules were used instead, the described modifications should be computed at run-time and, as it will be described in the following sections, they are very computationally intensive. Hence, they may incur into unaffordable run-time delays.

IV. Early-fetch and Reliability Improvement

A. The Scoring Function

In addition to the condition defined in Eq. (1), in order to decide if task τ is an appropriate candidate to apply the *Early-fetch* technique, a scoring function has also been defined:

$$Scoring\ Function\ \ (\tau) = CHI_\tau - CRI_\tau - TRI_\tau \quad (3)$$

where:

1) CHI_τ (*Configuration Hidden Impact*): This metric evaluates how much the task graph makespan is reduced if Task τ has no configuration delay.
2) CRI_τ (*Configuration Residency Impact*): This metric evaluates how much the makespan increases if Task τ is early-fetched at right side of the previous task graph, and therefore its configuration data reside in the FPGA until τ starts its execution.

3) TRI_τ (*Task Replication Impact*): This metric evaluates how much the makespan of the task graph is increased if the redundancy level of Task τ increases by one.

Tasks with $Scoring\ Function\ (\tau) \geq 0$ that also meet Eq. (1) are considered as candidates for early-fetch. The objective of this technique is to improve the reliability (R_{TG}) and Mean Time to Failure ($MTTF_{TG}$) of the task graph. These metrics will be further elaborated in the next subsection.

B. Reliability and Fault Model

In this study, the failures induced by soft errors, and in particular, by SEUs, are the object of concern. As indicated by [41], different altitudes above the Earth surface have different Soft Error Rates (SERs). In this paper, the reliability model presented in our previous work [42] has been used to estimate the reliability of a hardware task τ (denoted as R_τ). R_τ is the probability that the task executes from its start time to its finish time without any failure, with the condition that the task had no error when starting its execution.

This model assumes that at most one SEU occurs at a time, but one or more upsets might occur during task execution. Soft errors follow the Poisson distribution and they can be regarded as independent and random statistical events. Thus, the probability of a SEU in the sensitive bits of task τ, occurring j times, can be obtained as:

$$P\left(F_{\tau,j}\right) = e^{-\mu_\tau} \frac{{\mu_\tau}^j}{j!} \tag{4}$$

where

$$\mu_\tau = \rho \times (TS_\tau \times SB_\tau) \times (CT_\tau + RT_\tau) \tag{5}$$

in which ρ is the SER expressed in *#SEUs per bit per time* unit [41], TS_τ is task size in configuration memory, SB_τ indicates the percent of sensitive bits of Task τ [43], CT_τ is task computation time, and RT_τ is residency time of Task τ, indicating the time elapsed from when it is configured until it starts its execution. As this shows, despite the prefetch techniques increase the system performance, they also increase the probability of upsets in the task, which leads to the reliability degradation. The SER can be estimated by some modeling tools such as CREME96 [44].

Let $P\left(F_\tau\right)$ indicate the probability of failure of task τ given j SEUs during task execution, j ranging from 1 to ∞. Therefore we have:

$$P\left(F_\tau\right) = \sum_{j=1}^{\infty} P\left(F_{\tau,j}\right) \tag{6}$$

By having $P\left(F_\tau\right)$, the reliability of task τ is obtained as:

$$R_\tau = 1 - P\left(F_\tau\right) \tag{7}$$

In this work, it is assumed that an active redundancy-based FT technique is used for increasing task reliability [40]. With this technique, by replicating Task τ for r times, using the $1 - out - of - r$ scheme, the reliability of the fault-tolerant task τ_{ft} is given by [45]:

$$R_{\tau_{ft}} = \sum_{k=1}^{r} \binom{r}{k} (R_\tau)^k (1 - R_\tau)^{r-k} \tag{8}$$

Hence the reliability of task graph TG, after applying FT techniques, is obtained as [26]:

$$R_{TG} = \prod_{\tau_{ft} \in TG} \left(R_{\tau_{ft}}\right) \tag{9}$$

Finally MTTF of the task graph is calculated as inversely proportional to the task-graph probability of failure [43]:

$$MTTF_{TG} = \frac{MS_{TG}}{P(F_{TG})} = \frac{MS_{TG}}{1 - R_{TG}} \tag{10}$$

Where MS_{TG} is the makespan of TG.

This reliability model has been validated and discussed in more detail in [42]. In spite that it assumes that only SEUs can occur, this is a simplification that many authors make in their assumptions [43]. However, it would be easy to extend this model to k-bit Multiple-Cell Upsets (MCUs), since for each multiplicity k, their value of $P\left(F_{\tau,j}\right)$ would be calculated exactly as in Eq. (4), but with a different value for the SER (ρ). It is even possible to model the occurrence of MCUs and SEUs altogether, but the demonstration is too long to be included in this paper. In addition, any other reliability estimation methods (analytical, fault-injection, accelerated radiation tests...) can be used instead [46], [47], since they would be completely orthogonal to the methodology that this paper presents.

C. The Proposed Early-fetch Technique

The motivational examples of Section III have compared two possible modes of application of the proposed *Early-fetch* technique between the involved Stages $i-1$ and i.

1) Some modifications are carried out in just $RS\left(TG\left(i-1\right)\right)$ and $LS\left(TG\left(i\right)\right)$ (Figure 3), where $TG\left(i\right)$ indicates the task graph executed at Stage i. As a consequence, the early-fetch between Stages $i-1$ and i does not impact the applicability of this technique between Stages i and $i+1$.
2) Some modifications are carried out in $RS\left(TG\left(i-1\right)\right)$, $LS\left(TG\left(i\right)\right)$ and $RS\left(TG\left(i\right)\right)$ (Figure 4). In this case, due to the modifications introduced in $RS\left(TG\left(i\right)\right)$, the early-fetch between these two stages does impact the applicability of the technique between Stages i and $i+1$.

In the aforementioned examples, it was assumed that the same task graph is executed twice in the system. However, it is clear that, if another different task graph TG_j runs at Stage 2 (both in Figure 3 and Figure 4), the modifications carried out at the schedules of both task graphs could be completely different. Without losing generality, if n task graphs can be executed after the task graph of Figure 1, n different pairs of modifications can be introduced at the schedules of the involved task graphs.

In this study, in order to apply the *Early-fetch* technique, the profiling of all the n task graphs has been carried out at design time in order to obtain the modified versions of their schedules. At run-time, the proper version will be dynamically selected depending on the run-time conditions. In the previous case 1), the profiling of $TG\left(i\right)$ involves examining all the n task graphs that may run at Stage $i-1$. However, for the previous case 2), such profiling would involve considering

the complete sequence of task graphs at Stages $[1 \ldots i-1]$. The reader will quickly understand that, given the potentially large number of task graphs and stages that may exist in an actual system, in the latter case, such profiling is absolutely unfeasible since it would involve a combinatorial explosion of combinations. Therefore, the early-fetch has been restricted to what is depicted in Eq. (1) and Figure 3. In other words, *only tasks whose execution does not go further than Boundary can be candidates to be early-fetched*.

Thus, given a set of n task graphs (TGS) that can be executed in the system, the methodology presented in this paper carries out a $n \times n$ design-time profiling for each task graph $TG_x \in TGS$ in order to modify the initial schedules of all the possible pairs $RS\,(TG_x)$ and $LS\,(TG_y)$, $\forall TG_x, TG_y \in TGS$, by selecting the most appropriate task(s) from TG_y to be early-fetched in TG_x, assuming that TG_y runs immediately after TG_x.

In the examples of the previous section, the value of boundary was set to $LastConfigTime$. However, it was also stated that this value could actually be selected such that: $LastConfigTime \leq Boundary < Makespan$. The question that arises is: How to select the most appropriate value for this parameter? In the example of Figure 4, the only two tasks that are candidates for early-fetch are τ_1 and τ_2, since they are the two only ones whose execution time falls entirely within $LS\,(TG)$. However, if boundary was set to 806 *ms* (i.e., the end of execution of Task τ_3), then τ_3 would also be eligible for early-fetch, but it has the cost of reducing the time margin of $RS\,(TG)$ from 335 *ms* to 300 *ms* to early-fetch tasks from the next stage. In order to achieve a good trade-off between these two metrics, in the presented approach, this parameter has been set as follows:

$$Boundary\,(TG) = max(LastConfigTime\,(TG)\,, \quad Makespan\,(TG) - MaxConfigDelays) \tag{11}$$

where

$$MaxConfigDelays = \max_{TG_i \in TGS} ConfigDelay\,(TG_i) \tag{12}$$

and

$$ConfigDelay\,(TG_i) = \sum_{\tau_j \in TG_i} ConfigDelay\,(\tau_j) \tag{13}$$

which $ConfigDelay\,(\tau_j)$ indicates the configuration delay of task τ_j in the target device.

The complete approach is described in Algorithm 1. First of all, in the proposed algorithm the scoring function of the tasks of TG_y is calculated in Lines 2-8. In the next lines (Lines 9-11), each candidate task to be early-fetched is examined to obtain its MTTF difference (δ_{MTTF}) when applying this technique. Afterwards, tasks are sorted decreasingly by their δ_{MTTF} (Line 12). Then, the algorithm calculates the time elapsed between boundary and the makespan of the previous task graph execution (TG_x), which is referred to as *FreeTime* (Line 13). This time will be used to know how many tasks from the current task graph (TG_y) can be early-fetched in the previous one (TG_x). The candidate tasks to be early-fetched are selected according to their δ_{MTTF}. At each iteration, it is assessed if each candidate task τ_i can be early-fetched within the *FreeTime* of the previous stage, and if its additional replica can be added in the current one (Line 15). If this condition is true, τ_i is early-fetched, then the right side of the previous schedule, the left side of the current one and *FreeTime* are updated (Lines 16-19). The algorithm returns these two new subschedules for $RS(TG_x)$ and $LS(TG_y)$ (Line 22).

Algorithm 1 The Proposed Early-fetch and Reliability Improvement Approach

1: **input** $\text{TG}_x,\ \text{TG}_y$; *// Order of execution: TGx (Stage i), TGy (Stage i+1)*
2: **for all** tasks $\tau_{i,y}$ **in** TG_y **do**
3:     **if** (FinishTime($\tau_{i,y}$) $\leq$ Boundary(TG_y)) **then**
4:         $SF_{\tau_{i,y}}$ = Scoring Function($\tau_{i,y}$)
5:     **else**
6:         $SF_{\tau_{i,y}} = -1$
7:     **end if**
8: **end for**
9: **for all** tasks $\tau_{i,y}$ **in** TG_y **with** $SF_{\tau_{i,y}} \geq 0$ **do**
10:     $\delta_{MTTF\,\tau_{i,y}} = MTTF_{new\,\tau_{i,y}} - MTTF_{old\,\tau_{i,y}}$
11: **end for**
12: **sort** tasks in a decreasing order of δ_{MTTF}
13: FreeTime(TG_x) = Makespan(TG_x) $-$ Boundary(TG_x)
14: **for all** sorted tasks $\tau_{i,y}$ **in** TG_y **with** $SF_{\tau_{i,y}} \geq 0$ **do**
15:     **if** (FreeTime(TG_x) $\geq$ ConfigDelay($\tau_{i,y}$) **and** increasing the redundancy level of $\tau_{i,y}$ does not violate the time and size limitations) **then**
16:         **EarlyFetch** $\tau_{i,y}$ **in** TG_x and **increase** its redundancy level
17:         RS_{new} (TG_x) = **update** Schedule(TG_x, Right)
18:         LS_{new} (TG_y) = **update** Schedule(TG_y, Left)
19:         FreeTime(TG_x) = FreeTime(TG_x) $-$ ConfigDelay($\tau_{i,y}$)
20:     **end if**
21: **end for**
22: **return** RS_{new} (TG_x), LS_{new} (TG_y)

D. Hardware Implementation

For each pair of task graphs TG_x and $TG_y \in TGS$, such that TG_y is executed immediately after TG_x, the result of the profiling described in the previous subsection is a pair schedule versions: one for $RS(TG_x)$ and another one for $LS(TG_y)$. Therefore with n task graphs in the system, $n \times n$ schedule versions are generated at design time for each task graph. At run-time, the proper ones are selected dynamically, depending on the run-time sequence of running task graphs. This is illustrated in Figure 5, where one can see that $n+1$ versions of $LS(TG_j)$ and another $n+1$ versions for the $RS(TG_j)$ are possible (the n generated schedule versions plus the by-default one). In case no information exists at run-time about the previous or next task graphs, the selected schedule is just the original one (this is indicated in the figure by means of the symbol $\emptyset$). This happens, for instance, when a task graph is executed after a system reset; or when at the time a task graph finishes its execution, no other task graph is requested for execution yet (and hence, the system remains idle for a while).

In order to carry out the proper run-time selection of the task graph schedules in a transparent and efficient manner, this paper also presents a hardware architectural support (Figure 6) that can be implemented using some of the reconfigurable resources of the target FPGA. In our implementation, the

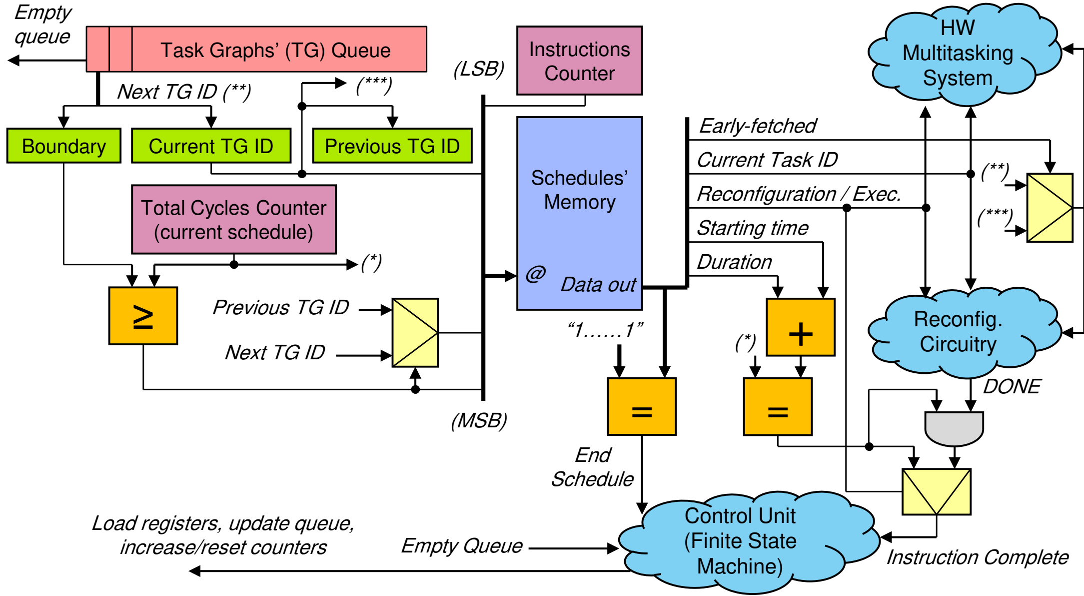

Fig. 6. Implementation details of the proposed technique with multiple task graphs

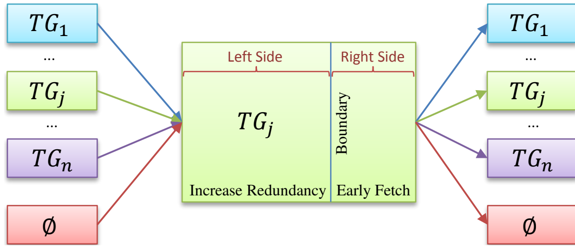

Fig. 5. Different versions for the schedule of Task Graph TG_j, depending on the previous and next task graph to be executed

pair schedules of all the possible task graphs are stored in a memory (see the figure). It is assumed that a schedule is composed of a set of instructions, each of which has the following information:

- *Task ID*: The ID of the task that is going to be scheduled. The ID of a task is unique among all tasks of the task graph.
- *Reconfiguration/Exec.*: Indicates if the task will be reconfigured, or executed. This field is just 1 bit ('1' = reconfiguration; '0' = execution).
- *Starting time*: The starting point of time, in terms of clock cycles, where the instruction (reconfiguration or execution) must take place.
- *Duration*: The time (in terms of the number of clock cycles) that the instruction under execution takes to be completed.
- *Early-fetched*: Indicates whether the task is early-fetched from the next task graph ('1') or not ('0'). This field is just 1 bit.

This information corresponds to the output data port of the Schedules' memory, where the instructions are read (see Figure 6).

The proposed system has been designed to work autonomously since the moment when the schedule of a task graph is requested. It features a queue of task graphs to be executed (*Task graphs' (TG) queue*), which has been implemented using a fixed First-In-First-Out (FIFO) approach. This architecture is assumed to communicate with an upper layer of middleware or an operating system that dispatches the task graphs at run-time.

When the queue is not empty, the system starts carrying out the proper scheduling operations assigned to the first task graph in the queue. The hardware described in Figure 6 is steered by a *Control Unit*, which has been implemented as a finite state machine. It implements the pseudo-code presented in Algorithm 2. Thus, if the task graphs' queue is not empty, the first step is to read the first task graph from the queue (Line 2). Two pieces of information are stored for a task graph: its unique *ID*, and its value for *Boundary*. Both of them are read from the queue and stored in separate registers in the architecture (see Figure 6). An additional register stores the ID of the task graph that was executed prior to the current one (*Previous TG ID* register). This register is used to select the appropriate schedule from the memory, as it was explained above.

With this information, the schedule of the current task graph is retrieved from the schedules' memory, instruction by instruction (Lines 3-7 in Algorithm 2). The address port of this memory is connected to the following 4 pieces of information, sorted from the Most Significant Bit (MSB) to the Least Significant Bit (LSB):

- One bit indicating if the instruction belongs to the left or the right side of the schedule. This is known by comparing the *Boundary* value with the total number of clock cycles that have elapsed from the starting of the current scheduling stage (which are stored in the *Total Cycles Counter*). If *Total Cycles Counter* $<$ *Boundary*, this bit is '0'; otherwise, its value is '1'. Hence, the lower half of the memory stores the left sides of all the

Algorithm 2 Implementation of the Control Unit

1: **while** (Task Graphs' Queue $\neq \emptyset$) **do**
2: TG = Read from queue();
3: **for all** $instructions_i$ **in** schedule (TG) **do**
4: $instruction$ = Read from memory ($instructions_i$);
5: **wait until** Total Cycles Counter == Starting time($instruction$) + Duration($instruction$);
6: $instructions\ counter$++;
7: **end for**
8: **update** Next Task Graphs' Queue;
9: **if** (Task Graphs' Queue $\neq \emptyset$) **then**
10: Previous TG ID register = Current TG ID register;
11: **else**
12: Previous TG ID register = $\emptyset$;
13: **end if**
14: **reset** counters;
15: **update** Task Graphs' Queue;
16: **end while**

schedules, whereas the upper half of the memory does likewise with the schedules' right sides.

- The ID from the previous (or the next) task graph to be executed. This information is retrieved from the *Previous TG ID* register, and from the data output port of the task graphs' queue, respectively, and it is selected by the multiplexer that can be seen in Figure 6. The selection signal of this multiplexer is the bit described in the previous paragraph.
- The ID of the task graph currently under execution (*Current TG ID* register).
- The output of a counter that keeps track of the schedule's instructions that have been executed so far (*Instructions Counter*).

This allows storing the information of the many possible schedules in the memory in a modular way: the instructions of each side (left or right) of the schedules are physically placed in adjacent positions in the memory, since the *Instructions Counter*'s output is connected to the LSBs of the memory address port. The exact location of these instructions in the memory is determined by the values of the IDs of the previous and next task graphs to be executed. Thus, this hardware support allows fetching the proper instructions in an automatic and transparent manner, with negligible delays and with low resources consumption. Thus, at run-time, depending if the total cycles counter is below or above the *Boundary* value, the instructions will be fetched from the lower or upper half of the memory, respectively, in a very simple but effective manner.

When an instruction is fetched from the memory (Line 4 in Algorithm 2), the signals *Current Task ID, Reconfiguration/Exec.* and the output of the multiplexer that is connected to the *Early-fetched* bit are transmitted simultaneously both to the hardware multi-tasking system that runs the tasks; and to the reconfiguration circuitry (depicted in the Figure 6). The latter multiplexer is used to select the ID of the task graph that the current task belongs to. Thus, if *Early-fetched* = 0, then the task belongs to the task graph indicated in the *Current TG ID* register. Otherwise, it belongs to the next task graph, which is indicated by the *Next TG ID* signal (in other words, it has been early-fetched).

Describing the reconfiguration circuitry and the HW multi-tasking system is out of the scope of this paper, since there are many implementation options for both of them available in the literature [28], [29], [39]. All of them assume that the available resources are divided into a number of partially reconfigurable regions that host the execution of the hardware tasks. That system is also assumed to manage the communications among tasks, as well as the correct execution of the tasks taking into account their FT technique [38]. In addition, it is assumed that the physical placement of the tasks has been decided elsewhere: the hardware depicted in this section only triggers the reconfiguration/execution of the tasks in the reconfigurable hardware, exactly on the location specified in the programming file of the task. This location has been decided at design time by the placer in another step of the flow.

The value of the *Total Cycles Counter* is used to compare if the current schedule's instruction has finished or not (activation of the signal *Instruction Complete* in Figure 6). Thus, in case *Reconfiguration/Exec.* = '0' (task execution), the following condition is checked:

$$Total\ Cycles\ Counter == Starting\ time + Duration \tag{14}$$

If this condition is true, the *Instruction Complete* line is activated, by selecting the result of the comparison with the multiplexer. In case *Reconfiguration/Exec.* = '1', an additional condition is checked: if the reconfiguration circuitry has finished carrying out the reconfiguration of the current task (by selecting the other input line of the multiplexer and the AND gate). In either of these two cases, while this condition is not true, the *Control Unit* increases the *Total Cycles Counter* by one and the same comparison is made again and again, cycle after cycle (Line 6 in the algorithm). When this condition finally becomes true, the *Control Unit* triggers the execution of the next schedule's instruction by increasing the *Instructions Counter* by one, then by reading the next instruction from the memory, and by repeating again the process. All this is equivalent to the iterations of the FOR loop in Algorithm 2.

It is important to highlight that the control word "111......111" is used to identify the end of the schedule of the *Current TG*. Thus, when the *End Schedule* line in Figure 6 is activated, the schedule finishes, the queue of task graphs and the *Previous TG ID* register are updated, and the two counters are reset (Lines 9-14). Note that the *Previous TG ID* register is updated to the value of *Current TG ID* only when, at that time, there is another task graph in the queue. Otherwise, it is updated to a null value. This is done in order to ensure that, if the *Current TG* was executed assuming that the following one is null, then the following one is also executed assuming that the previous one is also null; and vice-versa. Finally, the task graphs' queue is updated only after the execution of each task graph (hence, if there is a task graph whose execution is requested while another one is running, the queue will be updated only at the time instant marked by *Boundary*). Finally, when a task graph finishes its execution and the Task Graphs' Queue is not empty, the algorithm will run again, otherwise it waits until the next request of a task graph execution.

TABLE II
ESTIMATED SERS FOR DIFFERENT ORBITS AND SOLAR CONDITIONS

	SEUs/bit/Day for different solar conditions (Xilinx Virtex-5 XUPV5LX110T FPGA)			
Orbit	**Solar Max**	**Worst Week**	**Worst Day**	**Peak 5-Min**
GEO	$6.09{\times}10^{-8}$	6.47×10^{-5}	$3.35{\times}10^{-4}$	$1.29{\times}10^{-3}$
GPS	$6.09{\times}10^{-8}$	5.71×10^{-5}	$2.89{\times}10^{-4}$	$1.10{\times}10^{-3}$
MOL	$3.01{\times}10^{-7}$	6.09×10^{-5}	$3.12{\times}10^{-4}$	$1.18{\times}10^{-3}$
POL	$2.25{\times}10^{-7}$	1.33×10^{-5}	$7.99{\times}10^{-5}$	$2.97{\times}10^{-4}$
LEO	$9.51{\times}10^{-8}$	5.71×10^{-9}	$4.19{\times}10^{-9}$	$1.52{\times}10^{-8}$

V. EXPERIMENTAL RESULTS

A. Experimental Setup

In order to evaluate the proposed technique, several experiments have been done on actual task graphs obtained from multi-media applications. These task graphs are categorized in two groups:

- *Image Applications*: Two versions of the JPEG decoder (Serial and Parallel), an MPEG-1 encoder, and a pattern recognition application (HOUGH) [3].
- *Video Applications*: A 3D rendering application based on the open-source Pocket-GL library (Pocket GL (1)–Pocket GL (9)). This application contains 9 different task graphs with 2, 4, 5, and 6 consecutive tasks [3].

The model presented in Subsection IV-B has been employed to estimate the reliability of the tasks. For this purpose, different values for the SER have been used. As indicated by [48], different altitudes above the Earth have different SERs, which can be measured as *#SEUs per bit per time unit*. In order to have realistic estimations, we have used the SERs of the following four "harsh" orbits: Geosynchronous (GEO), Global Positioning System (GPS), Molniya (MOL), and Polar (POL). In addition a Low Earth Orbit (LEO) has been used as a point of reference as it features the lowest SER (see Table II). For each orbit, the SER is estimated for different solar conditions as: Worst Week, Worst Day, Peak Five Minutes, and Solar Max conditions of a Solar Energetic Particle (SEP) event [42], [47] for the Xilinx Virtex-5 XUPV5LX110T FPGA [49], using the CREME96 tools [44]. We believe that the estimations are reliable because the selected FPGA's technology has been largely studied in the literature against different sources of radiation [50], [51]. By using the documentation provided by the manufacturer and by carrying out experimental measurements, it was possible to calculate the reconfiguration overhead of tasks in this device.

In addition, the HW manager that was described in Subsection IV-D has been implemented on an FPGA. In this case, the Xilinx Virtex UltraScale XCVU095-2FFVA2104E FPGA has been used. We have selected that device for implementation because it is included in the UltraScale VCU108 evaluation kit, which is a prototyping board that includes the necessary elements to easily implement any hardware design, at a reasonable cost, on a state-of-the-art FPGA [52].

B. Performance Evaluation for Static Soft Error Rates

In the first experiment, task graphs are executed assuming that the SER does not change over the time. This experiment examines two cases: executing task graphs individually, and executing multiple task graphs altogether. The SER that is used in this experiment is the average value of the lowest (LEO – Worst Day) and the highest SERs (GEO – Peak 5-Min) that have been tabulated in Table II.

The results for individual task graphs have been presented in Table III. Task graphs' characteristics including task count, makespan, boundary value, and MTTF obtained by *ASAP* scheduling strategy have been depicted in the table. Then, the MTTF and the MTTF improvement of the task graphs, achieved by applying the proposed *Early-fetch* technique, have been shown. Finally, the last column shows the number of early-fetched tasks.

This experiment shows the positive impacts of applying the proposed technique to actual task graphs, so that without deteriorating their makespan, the MTTF has been improved by 114% on average. It is noteworthy to state that using other SERs yields very similar results in terms of MTTF improvement.

The proposed technique has also been applied to sequences of multiple task graphs. The obtained results have been presented in Table IV. This experiment examines three different groups of task graphs: Image Applications, Video Applications, and a combination of all the task graphs. In this experiment, for each set of task graphs, two different cases have been examined:

1) *Early-fetch*: The performance of the proposed technique. Let us remember that only the tasks that finish completely before the boundary are eligible to be early-fetched. Otherwise, as discussed above, the $n \times n$ task-graph profiling is unfeasible.
2) *Ideal Early-fetch*: An ideal scenario, where the run-time task-graph execution order is known in advance. In this case, a customized task-graph profiling has been made to obtain the modified schedules. In this case, all the tasks (even those finishing after boundary) were eligible to be early-fetched. For this experiment, a sequence of 100 random stages has been generated.

As the obtained results show, in this case the proposed *Early-fetch* technique has very positive impacts on the MTTF. In addition, these results show that the ideal case yields a MTTF improvement three or four times greater than the *Early-fetch* technique. The reason is that the MTTF improvement grows exponentially when the reliability of the task graphs approaches 1 (see Eq. (10)). In other words, the number of early-fetched tasks has an exponential impact on the MTTF improvement. Thus, for instance, when one task is early-fetched the MTTF improvement is, on average, +25%. When two tasks are early-fetched, this improvement becomes +110%; but when three tasks are early-fetched, +415% MTTF improvement is achieved.

TABLE III
REAL-WORLD TASK GRAPHS USED IN THE DEVELOPED EXPERIMENTS

Task Graph	Task Count	Makespan (*ms*)	Boundary (*ms*)	MTTF Basic (*ms*)	Improved MTTF (*ms*)	MTTF Improvement (%)	Number of Early-fetched Tasks
JPEG (Serial)	4	80	73	5.1×10^5	1.4×10^6	165.25%	3
JPEG (Parallel)	8	55	40	3.2×10^5	1.7×10^6	426.54%	7
MPEG1	5	39	32	6.3×10^5	1.5×10^6	141.04%	3
Hough	6	96	88	5.1×10^5	8.1×10^5	58.21%	3
Pocket GL (1)	2	7	4	2.1×10^6	3.2×10^6	50.00%	1
Pocket GL (2)	4	11	6	1.2×10^6	1.6×10^6	40.00%	2
Pocket GL (3)	4	35	29	7.4×10^5	1.5×10^6	101.13%	3
Pocket GL (4)	5	54	46	5.0×10^5	1.1×10^6	126.84%	3
Pocket GL (5)	5	11	7	1.1×10^6	1.4×10^6	28.57%	2
Pocket GL (6)	5	18	11	7.2×10^5	9.2×10^5	27.78%	2
Pocket GL (7)	5	33	26	5.9×10^5	6.9×10^5	17.14%	2
Pocket GL (8)	6	27	18	6.2×10^5	1.1×10^6	76.92%	3
Pocket GL (9)	6	72	63	4.5×10^5	1.5×10^6	220.29%	4

TABLE IV
EXPERIMENTS ON MULTIPLE TASK GRAPHS

Task Graphs Characteristics			Early-fetch		Ideal Early-fetch	
Task Graphs	Task Graph Count	Basic MTTF (*ms*)	Improved MTTF (*ms*)	MTTF Improvement (%)	Improved MTTF (*ms*)	MTTF Improvement (%)
Image Apps	4	4.9×10^5	1.0×10^6	104.48%	2.61×10^6	431%
Video Apps	9	9.0×10^5	1.3×10^6	47.55%	3.88×10^6	329%
All Task Graphs	13	7.7×10^5	1.2×10^6	54.77%	3.72×10^6	388%

C. Performance Evaluation for Dynamic Soft Error Rates

In the second experiment, the proposed technique has been evaluated under a dynamic SER environment. In this case, the aforementioned task graphs have been hardened with two state-of-the-art FT techniques, and then the *Early-fetch* technique has been applied to them. These two techniques are:

- *Adaptive Technique*: It is an adaptive FT technique, also known as "*Three-mode adaptive strategy*", which has been presented in [5]. It employs different FT techniques for different ranges of SERs, but in each SER, a specific FT technique is used for all the tasks. Thus, no redundancy is applied when the SER is lower than 10% of the expected range of SERs, Triple Modular Redundancy (TMR) is applied when the SER is above 50% of the expected range of SERs, and Duplication With Compare (DWC) is used otherwise.
- *Pareto-based Technique*: In a previous work [6], the authors have addressed the problem of applying optimal FT techniques to task graphs, w.r.t. a given schedule, using multi-objective optimization methods. The study has shown that it is possible to increase the MTTF of a task graph without deteriorating its makespan, by using some solutions of the Pareto-set obtained from the optimization method.

The obtained results have been illustrated in Figure 7. The experiments have been performed on SERs presented in Table II. The SERs have been categorized based on the *Adaptive* technique, but for the sake of clarity, for each SER category, a uniformly distributed subset of three of them has been evaluated. The obtained results show that the *Early-fetch* technique outperforms both the *Adaptive* and the *Pareto-based* techniques. In addition, the results demonstrate that the improvements achieved are much more significant over the *Pareto-based* FT technique in environments with lower SERs. Similarly as in the results shown in Table IV, the reason is that the MTTF increase is much faster when reliability closes to 1, and it reaches to infinite when reliability = 1 (Eq. (10)).

D. Hardware Implementation

Finally, the amount of hardware resources used for implementation of the proposed hardware architectural support is shown in Table V. This table shows the number of Look-Up Tables (LUTs) and Flip-Flops (FFs) used, and breaks it down into the different existing modules: The *Next Task Graphs' Queue*, the *Schedules' Memory* and the *Control Unit*. It can be observed that the amount of consumed resources is very affordable: no more than 0.03% of the total FFs and LUTs, whereas it instantiates 1.22% of the available BRAMs. The latter value is reasonable, taking into account that the system needs to allow for space to store all the schedule versions for all the task graphs, in all the possible scenarios that can exist at run-time.

These values refer to a Xilinx Virtex UltraScale XCVU095-2FFVA2104E FPGA [52]. These data correspond to a system with a maximum of 16 task graphs (hence, the number of bits to represent task graph ID, $n_{TG_{ID}} = 4$), at most 16 tasks in each task graph ($n_{task_{ID}} = 4$), schedules with up to 32

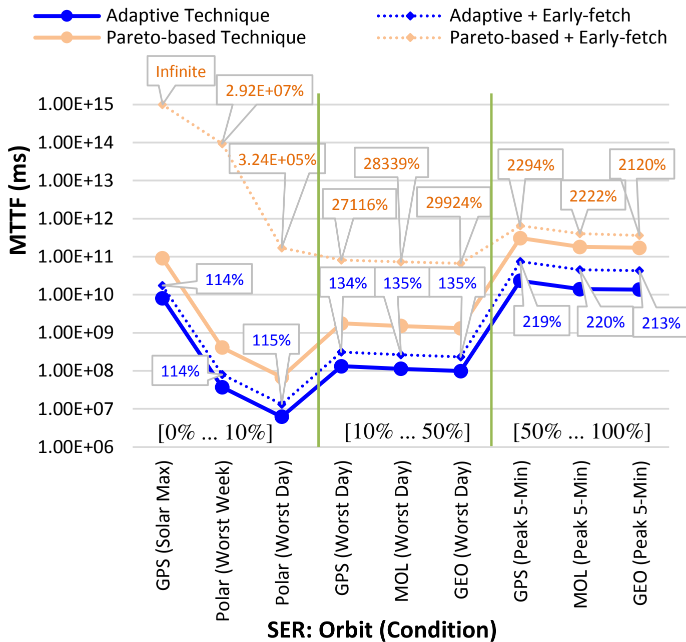

Fig. 7. MTTF improvement of the proposed Early-fetch technique over the FT approaches presented in [5] and in [6] (logarithmic scale)

TABLE V
RESOURCE CONSUMPTION OF THE HW ARCHITECTURAL SUPPORT

Module	CLB LUTs	CLB FFs	CARRY8	Block RAMs
Task Graphs' Queue	31 (<0.01%)	25 (<0.01%)	1 (<0.01%)	1 (0.03%)
Memory	43 (<0.01%)	1 (<0.01%)	0 (0%)	20 (1.16%)
Control Unit	98 (0.02%)	56 (<0.01%)	12 (0.02%)	0 (0%)
TOTAL	172 (0.03%)	82 (<0.01%)	13 (0.02%)	21 (1.22%)

different schedule instructions per side ($n_{instr} = 5$), a task graphs' queue with 256 positions and counters with a width of $n_{cycles} = 20$ bits. The latter parameter can be used to measure times for task reconfigurations and executions, for instance, ranging from 1 *us* to 1048.6 *ms* if the tasks' running clock frequency is 100 MHz. Thus, for the sake of simplicity, in this case, the width of the fields *Starting Time*, *Duration* (from memory data output), as well as those of the *Boundary* register and the *Total Cycles Counter* were set to the same value n_{cycles}.

This system scales well for different values of the parameters described above, but it must be taken into account that, every time the width of the memory's address port ($2 \times n_{TG_{ID}} + n_{cycles} + 1$) increases by 1, the amount of BRAMs that are needed doubles. Figure 8 and Figure 9 show the resources consumption for different values of this summation. As it can be seen, the FFs and LUTs consumption keeps under 0.12% in all the cases, but when $2 \times n_{TG_{ID}} + n_{instr} + 1 > 17$, the %BRAMs consumption reaches double digits. However, this is still an affordable cost for a system that supports a reasonably high number of different task graphs. Additionally,

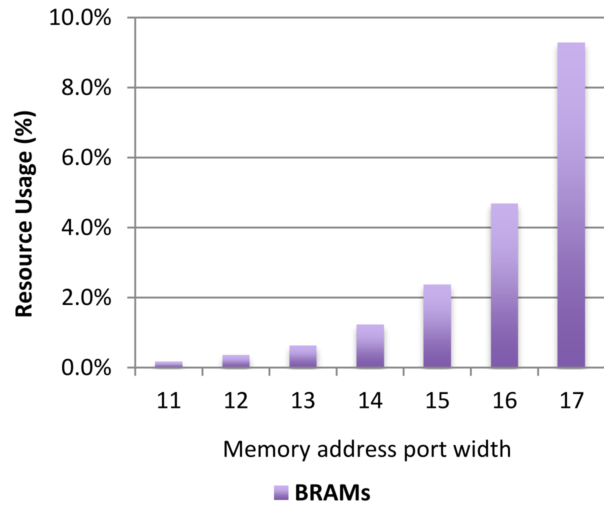

Fig. 8. %BRAM usage for different widths of memory address port

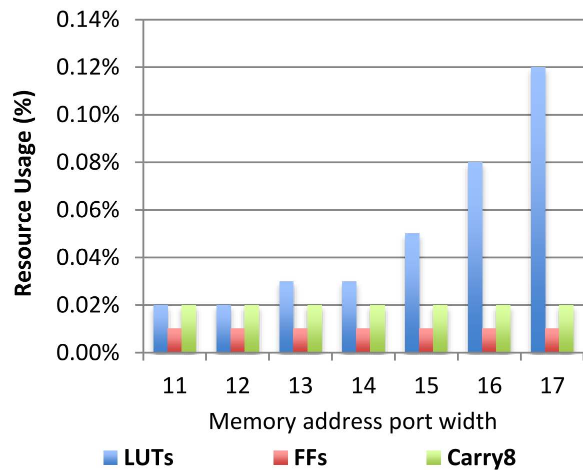

Fig. 9. %LUTs, FFs, and Carry8 usage for different widths of memory address port

if the length of the memory's output data port (see Figure 6 again); i.e., $2 \times n_{cycles} + n_{task_{ID}} + 2$, becomes greater than 64, then the total number of BRAMs doubles as well. Nevertheless, this does not happen unless task graphs with thousands of different tasks are used.

Focusing on the width of the output data port of the task graphs' queue; i.e., $n_{cycles} + n_{TG_{ID}}$, if this value becomes greater than 32, then the system will need 1 additional BRAM. Something similar happens when its depth is greater than 1024 positions. In these two cases, the increase of FF and LUTs consumption is negligible.

As discussed above, in this implementation, the bottleneck is clearly the embedded BRAMs consumption. Thus, if for instance, a small FPGA is used, a good solution would be to store the schedules in an off-chip memory (such as a FLASH, or a DDR2, commonly available in commercial FPGA-based prototyping boards), or a memory hierarchy composed of an on-chip cache plus an off-chip memory, which is very common in computer architecture. Of course, in this case, a cost in terms of performance loss has to be paid. However, even if

the performance of the proposed implementation decreased drastically, this would not involve significant run-time delays, since this system needs no more than 100 additional clock cycles to carry out the run-time computations. In addition, both this hardware and the multitasking system that steers the execution of the hardware tasks can work at different frequencies.

VI. Conclusions

This paper has presented a technique, named *Task Early-fetch*, to improve the MTTF of hardware applications represented as task graphs running on FPGA-based reconfigurable computers under harsh environments, without deteriorating their makespan. This technique receives as input a set of task graphs that can potentially run in the target system and, at design time, it applies two modifications to their schedules: On the one hand, it pre-fetches some tasks from a given task graph within the execution of the previous one. On the other hand, it increases the redundancy level of the selected tasks. Since the actual sequence of task graphs that will run in the system is not known at design time, this technique performs a $n \times n$ profiling, n being the number of task graphs. This paper has also presented a hardware architecture that carries out the proper run-time management of the modified schedules in an efficient and transparent manner, and with negligible run-time overheads.

The impacts of the proposed technique have been examined using a set of actual task graphs extracted from multimedia applications. Experimental results have demonstrated the positive effects of the proposed technique to improve the MTTF of hardware task graphs running on FPGA-based reconfigurable computers, in environments with static and dynamic SERs. Finally the low cost and the high performance of the presented prototype has been demonstrated.

References

[1] C. Bolchini, A. Miele, and C. Sandionigi, "Autonomous fault-tolerant systems onto SRAM-based FPGA platforms," *Journal of Electronic Testing*, vol. 29, no. 6, pp. 779–793, 2013.

[2] F. Say and C. F. Bazlamaçcı, "A reconfigurable computing platform for real time embedded applications," *Microprocessors and Microsystems*, vol. 36, no. 1, pp. 13–32, 2012.

[3] J. A. Clemente, J. Resano, C. González, and D. Mozos, "A hardware implementation of a run-time scheduler for reconfigurable systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 7, pp. 1263–1276, 2011.

[4] R. Velazco and F. J. Franco, "Single event effects on digital integrated circuits: origins and mitigation techniques," in *2007 IEEE International Symposium on Industrial Electronics*. IEEE, 2007, pp. 3322–3327.

[5] A. Jacobs, G. Cieslewski, A. D. George, A. Gordon-Ross, and H. Lam, "Reconfigurable fault tolerance: A comprehensive framework for reliable and adaptive FPGA-based space computing," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 5, no. 4, p. 21, 2012.

[6] R. Ramezani, Y. Sedaghat, M. Naghibzadeh, and J. A. Clemente, "Reliability and makespan optimization of hardware task graphs in partially reconfigurable platforms," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 53, no. 2, 2017.

[7] K. S. Morgan, D. L. McMurtrey, B. H. Pratt, and M. J. Wirthlin, "A comparison of TMR with alternative fault-tolerant design techniques for FPGAs," *IEEE Transactions on Nuclear Science*, vol. 54, no. 6, pp. 2065–2072, 2007.

[8] L. A. Tambara, F. Almeida, P. Rech, F. L. Kastensmidt, G. Bruni, and C. Frost, "Measuring failure probability of coarse and fine grain TMR schemes in SRAM-based FPGAs under neutron-induced effects," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2015, pp. 331–338.

[9] G. Cieslewski, A. Jacobs, C. Conger, A. D. George, and B. Kilpatrick, "Reconfigurable fault tolerance (RFT) for FPGA-based space computing," *Presentation for Military and Aerospace Programmable Logic Devices (MAPLD)*, 2008.

[10] E. Stott, P. Sedcole, and P. Y. Cheung, "Fault tolerant methods for reliability in FPGAs," in *2008 International Conference on Field Programmable Logic and Applications*. IEEE, 2008, pp. 415–420.

[11] S.-F. Liu, G. Sorrenti, P. Reviriego, F. Casini, J. A. Maestro, and M. Alderighi, "Increasing reliability of FPGA-based adaptive equalizers in the presence of single event upsets," *IEEE Transactions on Nuclear Science*, vol. 58, no. 3, pp. 1072–1077, 2011.

[12] O. A. Galashi, K. Mohammadi, and R. O. Gosheblagh, "Hybrid redundancy approach to increase the reliability of FPGA based speed controller core for high speed train," *Journal of Electronics (China)*, vol. 31, no. 3, pp. 256–266, 2014.

[13] N. H. Rollins and M. J. Wirthlin, "Reliability of a softcore processor in a commercial SRAM-based FPGA," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. ACM, 2012, pp. 171–174.

[14] L. Sterpone and M. Violante, "A new reliability-oriented place and route algorithm for SRAM-based FPGAs," *IEEE Transactions on Computers*, vol. 55, no. 6, pp. 732–744, 2006.

[15] K. Huang, Y. Hu, and X. Li, "Reliability-oriented placement and routing algorithm for SRAM-based FPGAs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 2, pp. 256–269, 2014.

[16] C. Bolchini, A. Miele, and C. Sandionigi, "A novel design methodology for implementing reliability-aware systems on SRAM-based FPGAs," *IEEE Transactions on Computers*, vol. 60, no. 12, pp. 1744–1758, 2011.

[17] F. L. Kastensmidt, L. Carro, and R. A. da Luz Reis, *Fault-tolerance techniques for SRAM-based FPGAs*. Springer, 2006, vol. 1.

[18] I. Herrera-Alzu and M. Lopez-Vallejo, "Design techniques for Xilinx Virtex FPGA configuration memory scrubbers," *IEEE Transactions on Nuclear Science*, vol. 60, no. 1, pp. 376–385, 2013.

[19] J.-Y. Lee, C.-R. Chang, N. Jing, J. Su, S. Wen, R. Wong, and L. He, "Heterogeneous configuration memory scrubbing for soft error mitigation in FPGAs," in *Field-Programmable Technology (FPT), 2012 International Conference on*. IEEE, 2012, pp. 23–28.

[20] H. Zhang, M. A. Kochte, M. E. Imhof, L. Bauer, H.-J. Wunderlich, and J. Henkel, "GUARD: Guaranteed reliability in dynamically reconfigurable systems," in *Proceedings of the 51st Annual Design Automation Conference*. ACM, 2014, pp. 1–6.

[21] J. Lach, W. H. Mangione-Smith, and M. Potkonjak, "Enhanced FPGA reliability through efficient run-time fault reconfiguration," *IEEE Transactions on Reliability*, vol. 49, no. 3, pp. 296–304, 2000.

[22] K. Elshafey, "Embedding fault tolerance via reconfiguration in configurable systems," in *Microelectronics, 2003. ICM 2003. Proceedings of the 15th International Conference on*. IEEE, 2003, pp. 370–373.

[23] C. K. Pang, A. Kumar, C. H. Goh, and C. V. Le, "Nano-satellite swarm for SAR applications: design and robust scheduling," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 51, no. 2, pp. 853–865, 2015.

[24] J. S. Hane, B. J. LaMeres, T. Kaiser, R. Weber, and T. Buerkle, "Increasing radiation tolerance of field-programmable-gate-array-based computers through redundancy and environmental awareness," *Journal of Aerospace Information Systems*, vol. 11, no. 2, pp. 68–81, 2014.

[25] S. Yousuf, A. Jacobs, and A. Gordon-Ross, "Partially reconfigurable system-on-chips for adaptive fault tolerance," in *Field-Programmable Technology (FPT), 2011 International Conference on*. IEEE, 2011, pp. 1–8.

[26] A. Das, A. Kumar, and B. Veeravalli, "Aging-aware hardware-software task partitioning for reliable reconfigurable multiprocessor systems," in *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2013 International Conference on*. IEEE, 2013, pp. 1–10.

[27] ——, "Reliability-driven task mapping for lifetime extension of networks-on-chip based multiprocessor systems," in *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 2013, pp. 689–694.

[28] T. Marconi, "Online scheduling and placement of hardware tasks with multiple variants on dynamically reconfigurable field-programmable gate arrays," *Computers & Electrical Engineering*, vol. 40, no. 4, pp. 1215–1237, 2014.

[29] I. Beretta, V. Rana, D. Atienza, and D. Sciuto, "A mapping flow for dynamically reconfigurable multi-core system-on-chip design," *IEEE*

Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 30, no. 8, pp. 1211–1224, 2011.

[30] J. Resano, D. Mozos, and F. Catthoor, "A hybrid prefetch scheduling heuristic to minimize at run-time the reconfiguration overhead of dynamically reconfigurable hardware," in *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*. IEEE Computer Society, 2005, pp. 106–111.

[31] J. A. Clemente, J. Resano, and D. Mozos, "An approach to manage reconfigurations and reduce area cost in hard real-time reconfigurable systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 4, p. 90, 2014.

[32] S. Hauck, "Configuration prefetch for single context reconfigurable coprocessors," in *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*. ACM, 1998, pp. 65–74.

[33] R. Cordone, F. Redaelli, M. A. Redaelli, M. D. Santambrogio, and D. Sciuto, "Partitioning and scheduling of task graphs on partially dynamically reconfigurable FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 5, pp. 662–675, 2009.

[34] F. Redaelli, M. D. Santambrogio, and D. Sciuto, "Task scheduling with configuration prefetching and anti-fragmentation techniques on dynamically reconfigurable systems," in *2008 Design, Automation and Test in Europe*. IEEE, 2008, pp. 519–522.

[35] J.-Y. Yin, G.-C. Guo, and Y.-X. Wu, "A hybrid fault-tolerant scheduling algorithm of periodic and aperiodic real-time tasks to partially reconfigurable FPGAs," in *Intelligent Systems and Applications, 2009. ISA 2009. International Workshop on*. IEEE, 2009, pp. 1–5.

[36] J. Yin, B. Zheng, and Z. Sun, "A hybrid real-time fault-tolerant scheduling algorithm for partial reconfigurable system," *Journal of Computers*, vol. 7, no. 11, pp. 2773–2780, 2012.

[37] R. Ramezani and Y. Sedaghat, "Scheduling periodic real-time hardware tasks on dynamic partial reconfigurable devices subject to fault tolerance," in *Computer and Knowledge Engineering (ICCKE), 2014 4th International eConference on*. IEEE, 2014, pp. 479–484.

[38] X. Iturbe, "Design and implementation of a reliable reconfigurable real-time operating system (R3TOS)," Ph.D. dissertation, University of Edinburgh, 2013.

[39] A. Agne, M. Happe, A. Keller, E. Lübbers, B. Plattner, M. Platzner, and C. Plessl, "ReconOS: An operating system approach for reconfigurable computing," *IEEE Micro*, vol. 34, no. 1, pp. 60–71, 2014.

[40] R. Ramezani and Y. Sedaghat, "An overview of fault tolerance techniques for real-time operating systems," in *Computer and Knowledge Engineering (ICCKE), 2013 3th International eConference on*. IEEE, 2013, pp. 1–6.

[41] J. Zhang, Y. Guan, and C. Mao, "Optimal partial reconfiguration for permanent fault recovery on SRAM-based FPGAs in space mission," *Advances in Mechanical Engineering*, vol. 5, p. 783673, 2013.

[42] R. Ramezani, J. A. Clemente, Y. Sedaghat, and H. Mecha, "Estimation of hardware task reliability on partially reconfigurable FPGAs," in *2016 16th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*. IEEE, 2016, pp. 1–4.

[43] J.-Y. Lee, Z. Feng, and L. He, "In-place decomposition for robustness in FPGA," in *2010 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2010, pp. 143–148.

[44] A. J. Tylka, J. Adams, P. R. Boberg, B. Brownstein, W. F. Dietrich, E. O. Flueckiger, E. L. Petersen, M. A. Shea, D. F. Smart, and E. C. Smith, "CREME96: A revision of the cosmic ray effects on micro-electronics code," *IEEE Transactions on Nuclear Science*, vol. 44, no. 6, pp. 2150–2160, 1997.

[45] M. L. Shooman, *Reliability of computer systems and networks: fault tolerance, analysis, and design*. John Wiley & Sons, 2003.

[46] O. Héron, T. Arnaout, and H.-J. Wunderlich, "On the reliability evaluation of SRAM-based FPGA designs," in *International Conference on Field Programmable Logic and Applications, 2005.* IEEE, 2005, pp. 403–408.

[47] P. S. Ostler, M. P. Caffrey, D. S. Gibelyou, P. S. Graham, K. S. Morgan, B. H. Pratt, H. M. Quinn, and M. J. Wirthlin, "SRAM FPGA reliability analysis for harsh radiation environments," *IEEE Transactions on Nuclear Science*, vol. 56, no. 6, pp. 3519–3526, 2009.

[48] G. Hubert and R. Ecoffet, "Operational impact of statistical properties of single event phenomena for on-orbit measurements and predictions improvement," *IEEE Transactions on Nuclear Science*, vol. 60, no. 5, pp. 3915–3923, 2013.

[49] Xilinx, "Virtex-5 FPGA Configuration User Guide, UG191(V3.10)," 2012.

[50] D. Hiemstra, P. Gill, and V. Kirischian, "Single event upset characterization of the Virtex-5 field programmable gate array Rocket IO using proton irradiation," in *Proceedings of the Xilinx Radiation Test Consortium Conference. Xilinx Inc*, 2010.

[51] H. Quinn, K. Morgan, P. Graham, J. Krone, and M. Caffrey, "Static proton and heavy ion testing of the Xilinx Virtex-5 device," in *2007 IEEE Radiation Effects Data Workshop*, 2007.

[52] Xilinx, "http://www.xilinx.com/products/boards-and-kits/ek-u1-vcu108-g.html," 2016.

Reza Ramezani was born in 1989. He received the B.Sc. and M.Sc. degrees in computer engineering from Shiraz Faculty of Engineering and Isfahan University of Technology, Iran, in 2010 and 2012, respectively. He is currently a Ph.D. candidate, with the Department of Computer Engineering, Ferdowsi University of Mashhad (FUM), Mashhad, Iran.

Since 2014, he collaborates with the Departamento de Computadores y Automatica (DACyA), at the Universidad Complutense de Madrid (UCM), Madrid, Spain. His main research area includes reconfigurable computing, real-time systems, task scheduling, and fault-tolerant designs.

Yasser Sedaghat received the M.Sc. and Ph.D. degrees in computer engineering from Sharif University of Technology, Tehran, Iran, in 2006 and 2011, respectively.

He is an Assistant Professor with the Department of Computer Engineering, Ferdowsi University of Mashhad (FUM), Mashhad, Iran. He has established and has been one of the chairs of the Dependable Distributed Embedded Systems (DDEmS) Laboratory, FUM, since 2012. His current research interests include dependable embedded systems and networks, reliable software design, embedded operating systems, and FPGA-based designs.

Juan Antonio Clemente received his computer science degree from Universidad Complutense de Madrid (UCM), Madrid, Spain, in 2007; and his Ph.D. in 2011. He is Assistant Professor and Researcher with the GHADIR research Group. Since 2012, he collaborates with the TIMA Laboratory, in Université Grenoble-Alpes, Grenoble, France.

His research interests are: dynamically reconfigurable hardware, FPGA design and task scheduling. Also, his research is focused on the study of Single Event Effects (SEE) tolerance of digital circuits implemented on FPGAs and he is being conducting experiments evaluating the robustness of memories face to neutrons and heavy ions with TIMA Labs and the Laboratoire de Physique Subatomique et de Cosmologie (LPSC), at CNRS-IN2P3 research center.