This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

RASA: Reliability-Aware Scheduling Approach for FPGA-Based Resilient Embedded Systems in Extreme Environments

Sangeet Saha, Xiaojun Zhai, *Senior Member, IEEE*, Shoaib Ehsan, *Senior Member, IEEE*, Shakaiba Majeed, and Klaus McDonald-Maier, *Senior Member, IEEE*

***Abstract*—Field-programmable gate arrays (FPGAs) offer the flexibility of general-purpose processors along with the performance efficiency of dedicated hardware that essentially renders it as a platform of choice for modern-day robotic systems for achieving real-time performance. Such robotic systems when deployed in harsh environments often get plagued by faults due to extreme conditions. Consequently, the real-time applications running on FPGA become susceptible to errors which call for a reliability-aware task scheduling approach, the focus of this article. We attempt to address this challenge using a hybrid offline–online approach. Given a set of periodic real-time tasks that require to be executed, the offline component generates a feasible preemptive schedule with specific preemption points. At runtime, these preemption events are utilized for fault detection. Upon detecting any faulty execution at such distinct points, the reliability-aware scheduling approach, RASA, orchestrates the recovery mechanism to remediate the scenario without jeopardizing the predefined schedule. Effectiveness of the proposed strategy has been verified through simulation-based experiments and we observed that the RASA is able to achieve 72% of task acceptance rate even under 70% of system workloads with high fault occurrence rates.**

***Index Terms*—Extreme environments (EEs), field-programmable gate array (FPGA), partial reconfiguration, real-time scheduling, reliability, resilient systems, single-event upsets (SEUs).**

I. Introduction

IN THE last couple of decades, field-programmable gate arrays (FPGAs) have found widespread use in embedded applications ranging from avionics to automotive, and object tracking [1] to cryptography [2]. Recent research [3], [4] shows that the FPGAs are also becoming increasingly popular in the field of robotics. Embedded systems within a robot provide advanced control techniques, environment reconstruction, interaction with humans, navigation in unknown and dynamic environments, etc. The execution of such scenarios often imposes stringent timeliness constraints and the robotic system must react within precise time interval to events in the environment [5], [6]. Hence, an FPGA-based embedded system for a robot demands well-defined scheduling methodologies, feasibility criteria, and admission control mechanisms for real-time task sets that are being executed.

Moreover, these embedded systems face reliability challenges when the robots are deployed in harsh environments, also known as extreme environments (EEs), as they become susceptible to errors due to severe conditions, such as high radiation levels, high temperature, etc. The deployment of robots for the exploration and inspection of an abandoned nuclear power plant is an example scenario of a robot system in EE. In such an environment, the operating FPGA inside the robotic systems might be affected by the charged radiation particles that strike the silicon of the FPGA [7] and it may cause *bit flips* in the architectural registers. This phenomena is commonly known as single-event upsets (SEUs) [7]. Radiation-induced SEUs can introduce errors within the FPGA logic and routing resources which inadvertently alter the device outputs. Radiation-hardened FPGA [8] provides an alternative way to mitigate this problem but it also imposes a huge cost overhead and performance constraints [9].

To leverage FPGA-based robotic systems in EE, the system must operate correctly and reliably even in the presence of transient faults. Fault-tolerant techniques, such as triple modular redundancy (TMR) [10] and duplication with compare (DWC) [10], have been employed to protect the system from most SEUs. However, TMR-based solution demands at least 200% area overhead for each replicated module [10]. Besides these methods, bitstream scrubbing [11], [12] techniques which exploit the reconfigurability feature of the FPGA are becoming popular, however, the improper use of such method may impose timing overheads. This essentially means that there is a need to develop an error detection and recovery (DAR) mechanism to ensure that no task deadlines are missed. Hence, the scheduler must have to incorporate a reliability measure while generating the schedule for the tasks set.

Recently, Saha *et al.* [13] have proposed a reliable scheduling strategy for the fully reconfigurable systems. However,

Manuscript received November 20, 2019; revised January 19, 2021; accepted April 24, 2021. This work was supported by the U.K. Engineering and Physical Sciences Research Council under Grant EP/R02572X/1, Grant EP/P017487/1, Grant EP/V000462/1, and Grant EP/V034111/1. This article was recommended by Associate Editor W. Shen. *(Corresponding author: Sangeet Saha.)*

Sangeet Saha, Xiaojun Zhai, Shoaib Ehsan, and Klaus McDonald-Maier are with the Embedded and Intelligent Systems Laboratory, University of Essex, Colchester CO4 3SQ, U.K. (e-mail: sangeet.saha@essex.ac.uk; xzhai@essex.ac.uk; sehsan@essex.ac.uk; kdm@essex.ac.uk).

Shakaiba Majeed is with the Real-Time Computing and Communications Lab, Hanyang University, Seoul 04763, South Korea (e-mail: shakaiba@rtcc.hanyang.ac.kr).

Digital Object Identifier 10.1109/TSMC.2021.3077697

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

in the case of fully reconfigurable systems, the main hurdle becomes the reconfiguration event which has to be synchronous for an entire FPGA and thus, consumes huge reconfiguration overhead and exhibits less efficiency. This is not suitable for real time and continuous operation as required by robots. To solve this problem, this work presents a methodology for the reliable scheduling technique for a set of safety-critical periodic tasks on runtime partially reconfigurable platforms. The proposed approach takes the advantage of partial reconfigurability feature of the system and attempts to increase the schedulability of tasks set in the presence of faults. We have coined our proposed approach as reliability-aware scheduling approach (RASA) and it follows a novel hybrid offline–online philosophy. The *offline component* generates a preemptive schedule for the periodic tasks by putting *preemption events/checkpoints* at appropriate intervals in such a way so that it can be effectively accessed by the online component. At runtime, we have illustrated how the offline generated schedule can be jeopardized by the radiation-induced faults. Upon detection of any unusual scheduling activity through our proposed detection mechanism, our *online component* will attempt to mitigate the faulty activity without hampering the pregenerated schedule.

The main technical contributions of this article are as follows.

1) An efficient preemptive, real-time, reliability-aware scheduling mechanism for partially reconfigurable FPGAs.
2) Analyzing the effect of radiation-induced faults on scheduling through a novel metric.
3) Low overhead online DAR approach to take remedial action in order to bring the system back to normalcy (without any deadline misses), in case of any faults.
4) Define the system's reliability through a "reliability cost" metric for the set of preemptive periodic tasks for specific intervals. This metric has also been utilized to measure the effectiveness of the proposed strategy.

Conducted simulation experiments with real-life parameters show the feasibility and the efficiency of the proposed approach. In particular, experimental results show that RASA is able to successfully schedule 72% of arrived tasks under high system load at 70%. Moreover, comparisons with existing research reveal that RASA comprehensively outperforms these methods (refer Section VIII).

The remainder of this article is organized as follows. Section II provides a brief discussion on related works. Section III discusses the system model used in this work. The proposed RASA strategy is presented in Section IV. The reliability analysis of the system is given in Section V. The analysis of the overhead of the proposed strategy is provided in Section VI. Section VII presents the experimental framework and results along with the discussion provided in Section VIII. Finally, conclusions are given in Section IX.

II. Related Work

There is a growing concern about the increasing vulnerability of modern embedded systems when they are employed in the harsh environments and thus, designing efficient resilient systems with fault-tolerant task handling strategies is the need of the hour. The generic problem of the fault-tolerant scheduling real-time tasks has branched out in different directions primarily based on: 1) types of underlying platforms; 2) fault model; and 3) detection mechanism. Now, we will make a close observation on each of these directions.

Variation in Platforms: Scheduling a set of real-time tasks in a multicore processor is an NP-hard problem and thus, various heuristic solutions have been employed to overcome this complication [14], [15]. In order to introduce fault-tolerant mechanisms within the scheduling strategy, hardware and time redundancy remain the most popular techniques in time-constrained systems [16], [17]. "Task replication" and "stand-by-sparing" [17] strategies are common *hardware redundancy* methods but, they impose considerable cost, specifically in EEs [16]. As an alternate approach, *time redundancy*-based solutions, such as "re-execution" and "recovery with checkpoint" [18], are challenging for the real-time systems subject to the stringent deadlines.

The problem of real-time task execution on FPGA-based systems has garnered considerable attention from the research community in recent years. Discussions on important works and research directions in this area may be found in [19]–[22]. Implementation of nonpreemptive online scheduling schemes like shortest job first (SJF) and first-come–first-serve (FCFS) for reconfigurable devices have been investigated in [23]. However, it is worth mentioning that high resource utilization cannot be achieved until the scheduling scheme is confined to its nonpreemptive nature. Poor resource utilization can be observed for nonpreemptive scheduling strategies if the deadlines or their utilizations are skewed and it can become worse if the number of processing elements increases [24].

Though preemptive scheduling techniques are powerful solutions for achieving high resource utilization, the research related to implement such techniques on FPGAs is still in its infancy. This is mainly due to the challenges involved in realizing the *hardware task context switching* on FPGAs, which literally refers to methodology on saving the execution state/status of the preempted task and reinitiate its execution from the saved state. However, Happe *et al.* [25] have shown that the hardware context switching can be realizable by *bitstream read back* methodology through an internal configuration access port (ICAP) on Virtex FPGAs.

Regarding fault-tolerant mechanisms in FPGAs, researchers have delved toward exploring spatial/hardware redundancy such as TMR [10]. Jacobs *et al.* [10] applied techniques, such as TMR and DWC, to schedule real-time tasks under various fault occurrence scenarios. Though these approaches consumed additional resource overhead, they did appear to be useful under low fault rates. We have compared the performance of our proposed RASA with these two strategies in Section VIII.

Besides TMR and DWC, bitstream scrubbing [11] is often used to correct errors after the FPGA suffered from SEUs. Sari *et al.* [26] combined scrubbing solution with a checkpoint/rollback mechanism in order to facilitate a low-cost solution for fault-tolerant scheduling. However, these methods have additional timing overhead due to periodic reading of

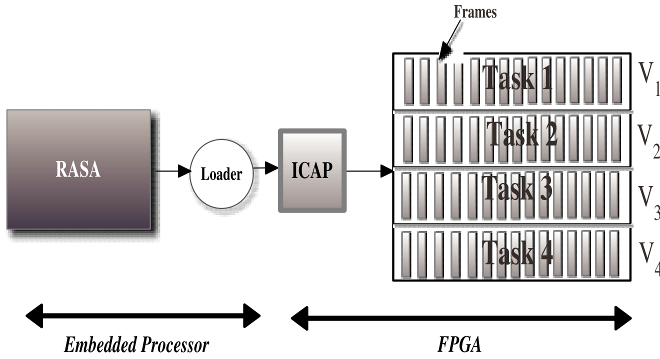

Fig. 1. Architecture model.

the configuration memory (CM). This article considers fault-tolerant real-time task scheduling on FPGAs and adopted a modified scrubbing mechanism at distinct preemption events of the real-time task set. Psarakis and Sari [27] showed how system utilization of FPGA-based real-time systems can be increased by using the "scrubbing-based" fault tolerance technique. This work employed EDF scheduling for handling the real-time tasks. We have also found this strategy proposed by Psarakis *et al.* to be an appropriate state of the art and thus experimentally compared our algorithm with this strategy.

Variation in Fault Model: In EEs, such as nuclear sites, radiation-induced faults in FPGAs mainly appear as transient faults [9]. Recent research has considered single and multiple transient faults. In [9] and [28], both in multiprocessor and FPGA domain, the Poisson distribution function is considered as a good estimation of fault occurrence. The fault occurrence rate (λ) also plays a crucial role in the fault model. This occurrence rate depends upon various environmental and operational factors. In EEs, for the old technologies, the fault rate lies between 10^{-2} and 10^{2} per hour [28]. However, as the modern day complex technology mainly based upon increasing growth of transistor size, increasing numbers of core and thus, attributed to the higher rate of faults. Recently, Haque *et al.* [29] have introduced the *fault burst* model, where they assume that multiple faults are occurring within a certain interval. However, in EEs, the occurrence of faults in "burst" is considered exceptional [30].

Fault-tolerant real-time systems mainly assume that the transient fault affects only the task running on a specific core rather than other cores [28]. Earlier it was assumed that any fault appeared in the specific core of a processor or in the FPGA fabric corrupts the output of the application. However, Burns *et al.* [31] have shown that these transient faults may also cause the over or under execution of tasks by executing them repeatedly or make them halt at undesirable instant, respectively. This article has considered multiple transient faults occurring at different processing elements by following Poisson distribution and each transient fault hampers the execution of a task.

Variation in Fault Detection: The fault detection mechanism plays a key role in designing a reliable and fault-tolerant system. Depending on the underlying platform, several methods for fault detection have been proposed. These methods can be broadly categorized into hardware and software components of a system [30]. The most popular mechanisms include: 1) sanity check of the outcomes at user level; 2) memory range violation or illegal instruction execution at OS level; 3) if the tasks are represented as control flow graph then detect the error flow; and 4) hardware duplication with comparison at hardware level.

Each of the above-mentioned methods only detects a specific type of fault; no technique is capable of handling any arbitrary fault. In this work, we have assumed that faults are detected at the end of each task or at a specified checkpoint using sanity checks.

III. System Model

In this section, we formalize the adopted architecture, task model, and fault model with the fault tolerance mechanism.

A. Architecture Model

The adopted architecture model in this article consists of a dynamically reconfigurable FPGA platform, an embedded processor (EP), and memory. A repository in the memory element is used to store and maintain the task bitstream images. EP carries out all the hardware reconfigurations by loading bitstreams from the repository into CM of the FPGA through ICAP. The internal architecture of FPGA has been assumed to be similar to that of the Xilinx Virtex series of FPGAs [32]. These FPGAs consist of a two-dimensional array of configurable logic blocks (CLBs) with components, such as multipliers (MULs) blocks, block RAMs (BRAM), clocks (CLKs), I/O blocks (IOBs), etc. The floor area of the FPGA is equipartitioned into M numbers of tile, $V = \{V_1, V_2, \ldots, V_M\}$, referred to as partially reconfigurable region (PRR) each of these PRR can execute any task irrespective of their resource demands. The size of each PRR is an integral multiple of the minimum reconfigurable portion (this is FPGA specific; for example, such a smallest region is called a *frame* in Xilinx and consumes a height of 20 CLBs for Xilinx Virtex 5 FPGAs [32]). Moreover, each PRR can be dynamically modified by loading the respective bitstream file, while the other regions continue their normal operation.

Our architecture model is shown in Fig. 1. Here, the floor area of the FPGA is partitioned into four PRRs. In this figure, it can be observed that the RASA algorithm proposed next, runs on the EP and as per the outcome of the scheduling strategy, the "loader" module loads/extracts respective task bitstream on a particular PRR (reconfigures the PRR) through ICAP.

B. Application Model

Let us assume that our target application $\mathcal{A}$ consists of n persistent periodic, independent real-time tasks, i.e., $\mathcal{A} = \{T_1, T_2, \ldots, T_n\}$ arrives for possible execution in M equisized PRRs $V = \{V_1, V_2, \ldots, V_M\}$. The temporal resource demand for a task T_i is given by the tuple $<e_i, d_i>$, where e_i time slots[1]

[1] All time values have been expressed in terms of "time slots" throughout this article. The length of a time slot has been assumed to be 1 ms.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

denotes the execution requirement and d_i time slots denotes the deadline. Without loss of generality, we represented the real-time constraint as: "T_i must compute e_i number of instructions within d_i time slots." For lucid explanation of this proposed strategy, it is assumed that one instruction is completed in one time slot using standard FPGA clock frequency.

C. Fault Model

FPGAs consist of a matrix of CLBs, connected by routing resources, with additional storage and input/output elements. It also contains a CM for storing the user implemented design/data and control bits. Radiation-induced SEU in these devices may produce a large number of single-event effects (SEEs), which generate one or more bitflips in the internal FPGA storage elements.

CM is the largest FPGA memory element and is used to store the user design. In this article, the focus is on transient and intermittent faults that affect FPGA CM, which may produce permanent errors in the functional logic and routing resources. The occurrence of SEUs in the time domain is assumed to follow a poisson process with constant rate λ per PRR. As all the PRRs are of equal size hence, the fault arrival rate for all PRRs is assumed to be identical. As mentioned in [27] that SEUs in the CM are the dominant failure type in modern FPGAs. Thus, in this article, only these faults are considered. It has been assumed that any SEUs in EP registers (if any) or in the memory can be tackled by using low-cost error correction codes (ECCs) mechanism.

Effect on Real-Time Processing: SEU affects the CLBs by changing the value of lookup tables (LUTs) and it may also alter the register contents. As a result, it will change the *context* (the register values storing the intermediate execution states) of any running hardware task. Thus, it will jeopardize the scheduling strategy which is primarily preemptive in nature and depends upon "context switching." Context switching on FPGA involves two distinct mechanisms: 1) saving the current state of a partly completed task and 2) restoring saved states to resume the execution. State-holding elements, such as Flip-flops and LUT-RAMs of CLB, are responsible for storing the "contexts" of a hardware task and get affected in EEs. It is assumed that any SEU can upset the context and with such affected contexts (CLB's contents) our hardware scheduling scheme will malfunction. More specifically, the fault within a PRR affects only the task running on that PRR.

D. Fault-Tolerance Mechanism

Typical fault-tolerance mechanisms for scheduling schemes conjugates two phases, that is: 1) a *fault detection* phase and 2) a *fault repair or recovery* phase. In the detection phase, at specific scheduling instances, we conduct a *bitstream read-back* procedure [33], which will extract/capture the *context* of tasks that were executing in the PRR prior to preemption. By evaluating captured context, the current execution status of the individual running task will be verified.

Based upon the outcome of the verification, the detection phase will be followed by a "recovery" phase. This recovery phase will select a particular recovery action (discussion in detail in Section IV-B). This recovery action will be performed by the "loader" through *CM scrubbing* technique [27] in order to rectify any abnormality in the execution. This "scrubbing" will be performed by configuring the respective PRR of the FPGA by downloading a new bitstream. The main concept behind this recovery is to exploit the "partial reconfigurability" feature of the FPGA. Only a PRR with a faulty task will be repaired through the reconfiguration, while the other PRRs will continue their operation. Bitstream extraction/loading will employ the respective in-circuit configuration interface, such as the loader. We termed the event which includes both *detect/recovery* as DAR and the corresponding overhead associated with this event is denoted as O_{dar}.

In our adopted system model, DAR shall be performed for each task on a PRR at different instances. It is obvious that it is not possible to conduct DAR simultaneously for different PRRs due to the limitations of the unique configuration port, ICAP. The remainder of the sections of this article will detail how the proposed strategy can handle the ICAP constraints along with the detailed discussion on our DAR methodology and calculation of associated overheads.

IV. RASA Algorithm

RASA employs a hybrid offline–online approach to provide a time partition-based reliable scheduling approach for a set of n periodic tasks $\mathcal{A} = \{T_1, T_2, \ldots, T_n\}$ on an FPGA, with a floor area that has been equipartitioned into M PRRs. RASA maintains time distinguished by the deadlines of the tasks. The interval between any two consecutive deadlines [say, the ηth and $(\eta - 1)$th task deadlines] is referred to as *time window* TW_η, TWL_η time slots denote the length of the ηth time window TW_η and can be calculated using

$$\text{TWL}_\eta = d_\eta - d_{\eta-1}. \quad (1)$$

As it has been assumed that the tasks are persistent, periodic in nature and known at design time thus, it will be sufficient to generate the scheduling information and store it for one hyperperiod[2] $\mathcal{H}$ of the tasks in $\mathcal{A}$.

The *offline* component, which is detailed further in Section IV-A (*RASA-Offline*) generates schedules of the periodic tasks for the entire duration of their hyperperiod. Each schedule reserves the space for checkpoints at appropriate intervals in such a way that it can be efficiently accessed by the online component. The *online* component as detailed in Section IV-B (*RASA-Online*) conducts time window wise scheduling as per the generated offline schedule. At each reserved checkpoint within the time window, if any abnormal scheduling activity is detected, the RASA-Online employs the appropriate recovery mechanism. Thus, the scheduling objective remains to maximize resource utilization so that rejection rates for the tasks may be minimized in the presence of faults during the online task execution. Before discussing the working principle of the offline and online components in detail, we now present a pseudocode of the entire RASA in Algorithm 1.

[2]Hyperperiod is given by the least common multiple (LCM) of the deadlines /periods.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Algorithm 1: RASA

{Given: task set $\mathcal{A}$ and set of PRRs V;}
for each time-window $TW_\eta \in \mathcal{H}$ **do** ***Offline***
1. Generate Schedule for all tasks for each PRR;
2. Specify the checkpoint / DAR intervals;

for each time-window $TW_\eta \in \mathcal{H}$ **do** ***Online***
1. Execute tasks as per generated schedule;
2. At each DAR interval check for anomaly for each task;

If faults are detected ***then*** employ recovery mechanism;
Else
Continue task execution as per schedule;

A. RASA: Offline Component (RASA-Offline)

Each periodic task T_i in $\mathcal{A}$ has a stipulated execution rate demand defined by its *weight*, $\text{wt} = (e_i/d_i)$, where e_i denotes the execution requirement and d_i denotes its period/deadline. For any time window (TW_η) of duration TWL_η in $\mathcal{H}$, each periodic task T_j is allocated a workload quota (Qu_j^η time slots) proportional to its weight and it can be calculated as

$$\text{Qu}_j^\eta = \left(\lceil \text{wt}_j \times \text{TWL}_\eta \rceil\right) \quad \forall T_j \in \mathcal{A}. \tag{2}$$

It can be noted that within a time window (say, TW_η) as all the available M PRRs will operate in parallel hence, the total system-wide capacity for that time window can be found as $\text{TWL}_\eta \times M$. In order to obtain a feasible schedule, this system-wide capacity must compensate the sum of workload quota of all tasks, i.e., $(\sum_{j=1}^{n} \text{Qu}_j^\eta)$. Thus, a necessary condition for scheduling to be feasible within TW_η is

$$\sum_{j=1}^{n} \text{Qu}_j^\eta \leq \text{TWL}_\eta \times M. \tag{3}$$

All modern FPGAs impose the constraint of full reconfiguration before conducting partial reconfiguration. Hence, full reconfiguration overhead (O_{Frg}) will be consumed at the beginning of the time window for each PRR.

1) Task Allocation: RASA-Offline selects tasks and attempts to allocate them from first PRR V_1. However, the combined sum of task workload quota along with the extraction, DAR overhead (O_{DAR}) in a particular PRR, V_i should be less than the time slice interval TWL_η. Available slack AS_i^η of each PRR V_i for the ηth time window can be calculated as

$$\text{AS}_i^\eta = \text{TWL}_\eta - O_{\text{Frg}}. \tag{4}$$

According to our strategy, if the available slack (As_i^η) of any particular PRR V_i is able to execute half of the workload quota ($\text{Qu}_j/2$) of a task T_j then T_j will be allocated in V_i. Otherwise, the next immediate PRR V_{i+1} (provided $i < M$) will be the possible candidate for allotment. However, in the case where V_i is the last available PRR ($V_i = V_M$), then system capacity is exhausted and scheduling cannot proceed in the time window TW_η.

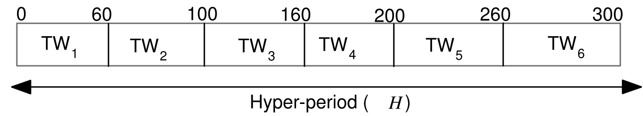

Fig. 2. Hyperperiod and time window.

2) Selection of Checkpoint/DAR: In order to make the scheduling approach reliable, we need to detect and repair any fault which occurs during the execution of the tasks. Thus, we have to judiciously select the number of DAR events for each given task such that the associated overhead does not jeopardize the deadlines. We have taken two approaches for this purpose and we will depict these approaches in two different cases.

Case 1: It has been assumed that if the following condition 5 becomes satisfied:

$$\text{Qu}_j^\eta \geq K \times O_{\text{DAR}} \quad \forall T_j \in \mathcal{A}. \tag{5}$$

DAR will be carried out after executing half of the allotted workload quota, i.e., $(\text{Qu}_j/2)$. After this allocation, the available slack will become

$$\text{AS}_i^\eta = \text{AS}_i^\eta - \left(\text{Qu}_j^\eta + O_{\text{DAR}}\right). \tag{6}$$

Note: We have empirically calculated K, the value of K lies in between 3 and 10 (see Section VII).

Case 2: In case Qu_j does not satisfy the condition stated in (5), there will be no intermediate check point. DAR will only be conducted after the task finishes its allotted workload quota. Similarly, the available slack can also be updated using (6).

The pseudocode for *RASA-Offline* is provided in Algorithm 2. For each time window, RASA-offline will calculate the workload quota. Provided the necessary condition stated in (3) is satisfied, the available slack for each PRR will be calculated. RASA-Offline then enters into a loop until all the tasks are scheduled, i.e., finish their allotted workload quota. Within the loop, RASA-Offline starts allocating tasks from the first PRR V_1, as per the task allocation strategy. Depending upon the size of the workload quota of a task (say, T_j), RASA-Offline places the checkpoint either in middle of T_j's execution or at the end of the execution of T_j. This approach is also illustrated with an example (Example 1).

Example 1: Let us assume a small hypothetical FPGA with $M = 4$ PRRs $V_1, V_2, \ldots, V_4$. The values of O_{Frg} and O_{DAR} are assumed as 6 and 2 ms. We have used these parameter values in all the worked-out examples presented in this article. Let us consider six real-time periodic applications $\{T_1, T_2, \ldots, T_6\}$ with the weights $(e_i/d_i)(28/60), (28/60), (25/100), (40/100), (14/60),$ and $(60/100)$. The length of the first time window $\text{TWL}_1 = 60$ (earliest task deadline = 60). The length of the second time window becomes $\text{TWL}_2 = 100 - 60 = 40$. The LCM of the two distinct task periods/deadlines becomes 300. Thus, the length of the hyperperiod ($\mathcal{H}$) is 300. The occurrence of TW_1 and TW_2 is shown in Fig. 2.

Algorithm 2: RASA-Offline

Input:
1.$\mathcal{A}$: The Application
2. M: Number of PRRs
3. e_i: The execution time
4. d_i: Deadline of a task
5. O_{Frg}: Full Reconfiguartion Overhead
6. O_{DAR}: DAR Overhead
7. TWL_η: Length of each individual time-window TW_η
8. $\mathcal{H}$: The hyper-period
Output:
Generate schedule for the application
for *each time-window* $TW_\eta \in \mathcal{H}$ **do**
  Calculate Qu_j^η for each task $T_j \in \mathcal{A}$ using equation 2;
  if equation 3 **NOT** satisfied **then** RETURN;
  Determine Available Slack (AS_i^η) using equation 4 for each PRR;
  while $\mathcal{A} \neq NULL$ **do**
    Initialise: $V_i = V_1$;
    if $\frac{Qu_j^\eta}{2} \leq AS_i^\eta$ **then**
      Select T_j for allocation in V_i;
      if $Qu_j^\eta \geq K \times O_{DAR}$ **then**
        Follow the approach stated in ***case 1***;
        Update Available Slack (AS_i^η) using equation 6;
        $Qu_j^\eta = Qu_j^\eta - (Qu_j^\eta/2)$;
      else
        Execute T_j as per the technique stated in ***case 2***;
        Update Available Slack (AS_i^η) using equation 6;
        Assign $Qu_j^\eta = 0$;
    else
      Move to PRR V_{i+1};
    Remove T_j from $\mathcal{A}$ if $Qu_j^\eta == 0$;

In this example, we have illustrated the task allocation performed by *RASA-Offline* for the first time window only. In this time window, the workload quota for each task can be found out by (2) and T_1 through T_6 will have workload quota as $\text{Qu}_1^1 = \text{Qu}_2^1 = 28$, $\text{Qu}_3^1 = 15$, $\text{Qu}_4^1 = 24$, $\text{Qu}_5^1 = 14$, and $\text{Qu}_6^1 = 36$. Fig. 3 shows the schedule generated for persistent periodic tasks by *RASA-Offline* in time window TW_1. After the allotment, we can observe that the V_1 has available slack (AS_1^1) 8 and similarly other PRRs have some amount of slack which varies from 3 (AS_3^1) to 34 (AS_4^1).

3) Avoidance of ICAP Conflict: All the modern FPGAs contain a single configuration port commonly known as ICAP. This ICAP port plays a crucial role while detecting and recovering the fault in the FPGA's CM. As there is only one ICAP, it is not possible to configure two PRRs simultaneously. Specifically, this implies that the DAR interval (O_{DAR}) cannot overlap on two distinct PRRs in time. RASA-Offline tackles any such circumstances as follows.

1) During the allocation of tasks on the first PRR, the instance of DAR is obtained in the course of allotments. Hence, the DAR instants are already known when the allocation of tasks for the second PRR begins.

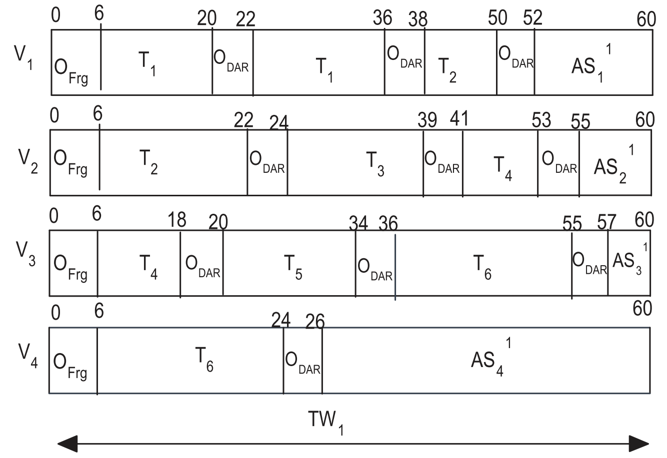

Fig. 3. Allocation of checkpoints and tasks by RASA-Offline.

2) If any DAR instant in the second PRR overlaps with the DAR instants in the first PRR, then the execution of task's workload quota in the second PRR are minimally adjusted such that the overlapping situation can be circumvented.
3) For example, we can observe in Fig. 3 that the workload quota of T_2 (Qu_2^1) is 14. So the specified DAR is about to take place at $t = 20$ and hence, the conflict appears with T_1 on first PRR. Thus, our adopted strategy postponed the DAR instance for T_2 on second PRR by two time slots,[3] i.e., at $t = 22$.
4) A similar approach should be taken while allocating tasks on the third PRR provided the DAR instances for the first and the second PRRs are already known. This approach will be followed till the last PRR in FPGA.

B. RASA: Online Component (RASA-Online)

Based on the schedule generated by *RASA-Offline*, we attempt to process our real-time tasks in our embedded system in EEs. The online component of RASA consists of DAR, i.e., the detect and recovery phase. We will now describe each phase in detail.

1) Detection Phase: As discussed in our fault model, the possible disruption in the FPGA's CM affects the behavior of the running tasks. As a consequence, it may be the case that all tasks are not able to complete their allotted course of execution states. In order to detect the current state of execution of a task, the *bit-stream readback* procedure will be employed at each DAR interval. Hardware task contexts are stored in state-holding elements, such as flip-flops, LUT-RAMs of CLBs, and other hard blocks (BRAMs, MULs, etc.). Thus, the "read-back" operation extracts the contexts of tasks that were executing in a PRR prior to a DAR event. These retrieved context, i.e., the register content will reveal the present status of task execution, it can be compared with the prestored register values (represents expected execution outcome). Through this comparison, it will be evaluated whether

[3] T_2 finishes its execution at $t = 20$ but the reconfiguration is only taking place at $t = 22$.

any task suffers from performance degradation due to a fault. We have formulated this "degradation" of task's performance through the following metric.

Let us assume that as per our *RASA-Offline* strategy, prior to any arbitrary DAR interval, task T_j has to complete I_j^i number of program steps/instructions (refer Section III-B) on PRR V_i. During the DAR, after reading back the bitstream, it may be found that T_j completes $\hat{I^i}_j$ instructions on PRR V_i. Hence, we can define a metric called "anomaly factor (AF)" for T_j on V_i as follows:

$$\mathrm{AF}_j^i = I_j^i - \hat{I^i}_j. \tag{7}$$

During each DAR event, AF_j^i will be calculated for each T_j to verify whether the task is hampered by the fault. However, there could be cases where no task is affected and it will be viable when $\mathrm{AF}_j^i = 0$. Hence, those task(s), which will exhibit $\mathrm{AF}_j^i > 0$, will be considered as affected and remedial actions will be taken for those tasks.

2) Recovery Phase: This phase will be initiated by evaluating the value of AF. If AF_j^i for any task T_j is found to be zero then it conveys that T_j has been executed satisfactorily and no recovery measure is needed. In those cases, where the value of AF becomes positive ($\mathrm{AF}_j^i > 0$), will infer that the respective task is not able to complete the desired number of instructions and expects the remedial action. At a particular DAR, based upon the value of AF and AS, our strategy will adopt different recovery actions as stated below.

As per the assumed model (discussed in Section III-B), workload quota Qu_j of T_j denotes that T_j has to complete Qu number of instructions within time window TW_η. Now, let us assume that in the time window (TW_η), at an arbitrary DAR interval, the following scenarios appeared for task T_j, executing on V_i.

1) *Scenario 1—$AF_j^i \leq AS_i^\eta$:* In this case, the degraded performance of T_j, which is quantified as AF_j^i, can be overcome by consuming the available slack (AS_i^η) within the TW_η.
2) *Scenario 2—$AF_j^i > AS_i^\eta$:* If the AF of T_j becomes such that it can no longer be compensated using the available slack of the PRR (V_i) then *RASA-Online* will take one of the following steps.
   a) Check other PRRs with maximum available slack. If there exists PRR V_j with available slack AS_j^η which can compensate AF_j^i then T_j will be executed on V_j.
   b) Otherwise, *RASA-Online* will speed-up the execution of T_j by enhancing the clock frequency. The frequency will be increased by a certain amount quantified rise-up factor (RUF) such that T_j can finish its desired execution by availing AS_j^η

$$\mathrm{RUF} = \frac{\mathrm{AS}_j^\eta}{\mathrm{AF}_j^i}. \tag{8}$$

Note: The clock frequency will be increased by enhancing the clock period from the dynamic clock management (DCM) system for that particular PRR only where the affected task is supposed to execute.

Algorithm 3: RASA-Online

```
Input: Time-window wise schedule generated by RASA-Offline
Output: Reliable time-window wise schedule where each task
        completes their alloted workload-quota
let t denotes the time-slot and initially, t = 0;
for each time-slot t do
    if t is bounday of a time-window (say, TW_η ∈ H) then
        Conduct full reconfiguration across all PRRs;
        Start executing tasks on PRRs in parallel, for the
          entire duration TWL_η as per RASA-Offline;
    else if t is a DAR instant within TW_η then
        Evaluate Anomaly Factor AF_j^i using Equation 7 for
          each task T_j on PRR V_i;
        if AF_j^i==0 then
            Execute T_j normaly as per pre-defined schedule;
        else
            if AF_j^i ≤ AS_i^η then
                Utilize AS_i^η to compensate AF_j^i;
            else
                Find max (AS_i^η)   ∀i ∈ M;
                {Say, V_j be the PRR having maximum
                  vailable slack, AS_j^η};
                if AF_j^i ≤AS_j^η then
                    Rectify the anomaly of T_j by executing on
                      V_j;
                else
                    Execute T_j on V_j by enhancing clock
                      using RUF;
    else
        Continue task execution;
        Simultaneously update Available Slack (AS_i^η   ∀i ∈ M)
          for all PRRs;
```

Algorithm 3 shows the pseudocode of our entire *RASA-Online* strategy. RASA-Online executes for each time slot t within a time window. At beginning of a time window, it conducts a full reconfiguration and starts task execution as per the schedule obtained from RASA-Offline. At each specified DAR instant (checkpoint), RASA-Online calculates the AF [see (7)]. Based upon the AF's value, an appropriate recovery action is selected as described in aforementioned scenarios 1 and 2. We will now illustrate the working principle of RASA-Online through an example for the same task set as described in Example 1.

Example 2: Let us consider the same scenario as discussed in Example 1. In the first time window (TW_1), schedule generated by *RASA-Offline* is demonstrated in Fig. 3. We will now depict the working principle of *RASA-Online* through each possible scenario and the entire scheduling scenario is depicted in Fig. 4.

Scenario 1: Let us assume that on PRR V_1 a fault occurs while T_1 is under execution after the first "DAR" interval and as a result, at the second DAR instant, i.e., $t = 36$, it has been found that T_1 has completed ten instructions. However, it should have completed 14 instructions (36–22). Thus, the AF

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

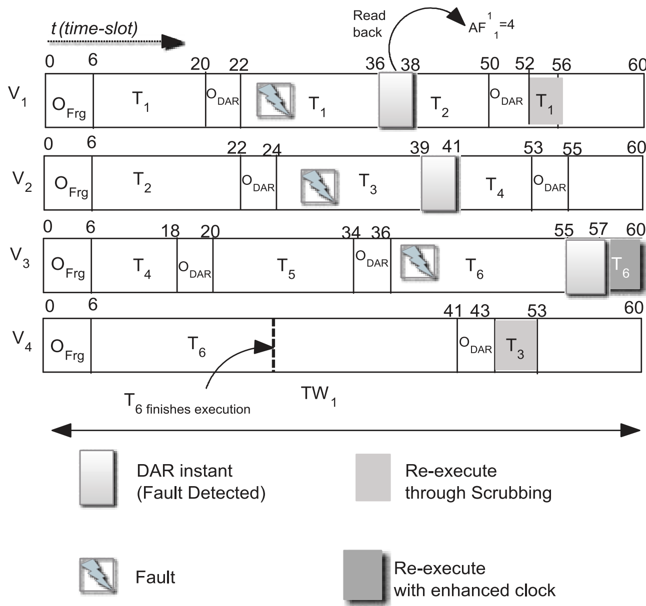

Fig. 4. Reliable scheduling via RASA-Online.

for T_1 on V_1 can be found as $\mathrm{AF}_1^1 = 14-10 = 4$. The available slack on V_1 is $\mathrm{AS}_1^1 = 8$. Hence, the degraded performance of T_1 can be circumvent by utilizing this slack and thus, T_1 completes its remaining four instructions (denoted by AF_1^1) through execution from 52 to 56 on V_1

Scenario 2: Let us consider that at the second DAR instant of V_2, a fault occurs, while T_3 is executing from time unit 24 to 39. At the second DAR instant, it is found that the execution of T_3 has been affected and is quantified as $\mathrm{AF}_3^2 = 10$. Now on PRR V_2, the available slack is 5 and it is thus not enough to compensate T_3. *RASA-Online* will execute T_3 on V_4, as it has the maximum available slack (see Fig. 4).

At DAR instant 55 on V_3, if it is found that T_6 has been affected by $\mathrm{AF}_6^3 = 6$. At $t = 57$, all the PRRs have an available slack of amount $60 - 57 = 3$ and thus, there is no such PRR which can accommodate re-execution of T_6. Hence, the only way the remaining execution T_6 (i.e., 6) can be compensated on a PRR (say, V_3) with available slack $\mathrm{AS}_3^1 = 3$, if the clock frequency can be enhanced by $\mathrm{RUF} = 2$ using (4), as shown in Fig. 8. At V_4, T_6 successfully completes at time instant 24.

V. Reliability Modeling

In this section, we attempt to address the issue of reliability for periodic tasks in reconfigurable platforms. Usually, reliability can be defined as the probability that the system can schedule an entire task set successfully, i.e., no tasks will miss its deadline. After analyzing the characteristics of periodic tasks, we will analyze the failure characteristic of our proposed model. A definition of reliability metric based on periodic, preemptive tasks set will then be investigated.

It should be noted that as the system deals with persistent periodic tasks, the particular schedule within the hyperperiod $\mathcal{H}$ will keep repeating. Also, the bit upset in the configuration frames can occur arbitrarily on a particular PRR V_i. These failures are assumed to be independent of each other and follow a Poisson process with a constant rate λ for all PRRs. Now, we will formulate the reliability of the system in the following steps.

Theorem 1: Reliability $(\Omega(\mathcal{H}))$ of the M tiled partially reconfigurable system executing persistent $\mathcal{A}$ periodic tasks with average $\mathcal{K}$ time windows within the hyperperiod $\mathcal{H}$ is

$$\Omega(\mathcal{H}) = \exp\left(-\sum_{\eta=1}^{\mathcal{K}}\sum_{i=1}^{M}\left(\sum_{j=1}^{n}\lambda_i \times \mathrm{Qu}_j^{\eta} \times \zeta_{ij}\right)\right) \quad (9)$$

where τ is the set of tasks running on PRR V_i, and $\mathrm{shr_j}$ denotes the share of an individual task $T_j \in \tau$.

Proof: The Poisson distribution is used for finding the probability of $\chi\,(\chi \geq 0)$ events occurred in a fixed time duration. We have assumed that events occur at a constant rate λ and their occurrences are independent. Thus, the probability of χ events during the length of a time window (say, TW_η) TWL_η is calculated as

$$\mathrm{Err}_\chi\big(\mathrm{TWL}_\eta\big) = \frac{e^{-\lambda TWL_\eta}\big(\lambda \mathrm{TWL}_\eta\big)^\chi}{\chi\,!}. \quad (10)$$

Let us assume that $\mathrm{NErr}_i(\mathrm{TWL}_\eta)$ is the probability that PRR V_i running without any failure ($\chi = 0$) within the time window. Therefore, $\mathrm{NErr}_i(\mathrm{TWL}_\eta)$ can be expressed as follows:

$$\mathrm{NErr}_i\big(\mathrm{TWL}_\eta\big) = e^{-\lambda_i \mathrm{TWL}_\eta} \quad (11)$$

where λ_i is the failure rate of V_i.

Hence, the reliability of the entire system (for all M PRRs) inside a time window TW_η can be written as

$$\begin{aligned}\Omega\big(\mathrm{TWL}_\eta\big) &= \prod_{i=1}^{M}\mathrm{NErr}_i\big(\mathrm{TWL}_\eta\big) = \prod_{i=1}^{M} e^{-\lambda_i \mathrm{TWL}_\eta}\\ &= \exp\left(-\sum_{i=1}^{M} -\lambda_i \mathrm{TWL}_\eta\right). \end{aligned} \quad (12)$$

Let us also assume that $\mathcal{K}$ is the number of time windows within the hyperperiod $\mathcal{H}$. Since the Poisson process is a steady incremental process, the reliability of a system can be investigated by measuring the reliability of a set of tasks running in the system within the hyperperiod of the task set. Thus, the reliability (Ω) of the M tiled partially reconfigurable system executing persistent $\mathcal{A}$ periodic tasks for $\mathcal{K}$ instances is given as follows:

$$\Omega(\mathcal{H}) = \exp\left(-\sum_{\eta=1}^{\mathcal{K}}\sum_{i=1}^{M}\left(\sum_{j=1}^{n}\lambda_i \times \mathrm{Qu}_j^{\eta} \times \zeta_{ij}\right)\right). \quad (13)$$

Here, element ζ_{ij} equals 1 if and only if T_j has been assigned to V_i; otherwise, $\zeta_{ij} = 0$. Inside a time window (say, TW_η) T_j will execute for Qu_j^η time slots (refer Section IV-A).

We are now in a position to determine the *reliability cost* of our proposed system with respect to a given set of persistent periodic tasks. The reliability cost of a task T_j on a PRR V_i for a time window can be defined as a product of failure rate λ_i of V_i and workload quota (Qu_j^η) of T_j. Thus, the reliability cost of the entire system within a time window is the summation over

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

reliability costs of all tasks assigned to that system (covering all PRRs) based on a given schedule.

Definition 1: Given a set $\mathcal{A}$ of real-time periodic tasks running on a partially reconfigurable system with a set of M PRRs, we compute the system reliability cost ($\Omega_{\text{cost}}(\mathcal{A}, M)$) of the system as

$$\Omega_{\text{cost}}(\mathcal{A}, M) = \sum_{\eta=1}^{\mathcal{K}} \sum_{i=1}^{M} \left(\sum_{j=1}^{n} \lambda_i \times \text{Qu}_j^{\eta} \times \zeta_{ij} \right). \tag{14}$$

It can be observed that while calculating the reliability cost (Ω_{cost}), the number of time windows within the hyperperiod ($\mathcal{H}$) has an important significance. However, the length of a time window can consume any arbitrary value depending upon the difference between two consecutive deadlines. However, in any case, if the length of the time window is too small and becomes close to the DAR interval, the functional scheduling cannot be performed. Hence, we have derived the following definition.

Definition 2: The length of an arbitrary time window (say, TW_η) TWL_η will be: $\text{TWL}_\eta = \max(d_\eta - d_{\eta-1}, \text{TWL}^{\text{th}})$. Here, TWL^{th} is a threshold value denoting the mandatory length of a time window and is an integral multiple of DAR interval.

VI. Quantification of DAR Interval, Overhead Analysis

This section provides a practical measure of the temporal overhead corresponding to a DAR event for a real-life FPGA. All the experiments have been carried out by considering this practical measure. In order to carry out our DAR mechanism, while reconfiguring any portion/region inside the FPGA, the currently executing task(s) on that region is brought to halt and then the corresponding bitstream is extracted/captured from the CM through ICAP.

It is evident from [33] that the actual context (register status) of a task remains within a small fraction (approximately about 8%) of its entire bitstream. Thus, at each DAR event, it will be sufficient to read-back only that part (which contains the actual context) of the bitstream to check the register status. This *selective read-back* mechanism drastically reduces the actual extraction overhead by (1/10)th amount compared to the read-back of the entire bitstream [33]. Following the extraction, the *detection* phase will evaluate the "AF" and based upon its outcome, the "recovery" phase will select a recovery action by executing Algorithm 2, RASA-Online. Then, the recovery action will be performed by loading a new modified bitstream in the CM, i.e., execute this scrubbing technique. Hence, in order to estimate the overhead of DAR, overheads associated to the following events should be considered.

1) *Detection:*
   a) Extracting information through *selective read-back*, carried out by the loader.
   b) Calculation of AF, carried out by the EP.
2) *Recovery:*
   a) Select appropriate recovery action using Algorithm 2, carried out by EP.
   b) Execute the recovery action through scrubbing, i.e., modified bitstream loading, carried out by the loader.

The temporal overhead corresponding to each individual event is estimated as follows.

Extracting Information Through "Selective Read-Back": The time required (say, TR) for extracting/restoring that is the loading of a particular region will be proportional to the size of the bitstream (Bi) being extracted/loaded and will be inversely proportional to configuration clock frequency (C_{clk}) of the loader and data bus width of the configuration port (DBW). Hence, TR can be expressed as

$$\text{TR} = \frac{\text{Bi}}{C_{\text{clk}} \times \text{DBW}}. \tag{15}$$

For a Virtex-5 ML507 board (XC5VFX70T), which has an area of 38×160 [32] CLB units, the full configuration bitstream is 3.37 MB [32], C_{clk} is 100 MHz and DBW = 32 bit [32], resulting in a TR of 8.5 ms as per (15). This value denotes the full extraction overhead for an entire FPGA. However, via the "Selective read-back" mechanism this *extraction overhead* can be reduced ($[1/10] \times 8.5$ ms $= 0.85$ ms ≈ 1 ms) in order to extract only the part which is our actual "region-of-interest."

AF Calculation and Recovery Action Selection: From the extracted context, the calculation of "AF" will be carried out by the associated EP and, based on the value, a single recovery action will be selected by using Algorithm 2. As the frequency of the associated EP of a Virtex-5 FPGA can be as high as 550 MHz [34], these steps will not be of significant computation cost. Nevertheless, we have pessimistically assumed that the overhead is equal to the 8.5 ms (the worst case total bitstream extraction time).

Execute the Recovery Action Through Scrubbing: Based upon the selected recovery action, the "loader" will now load a modified bitstream as a scrubbing technique. This loading time will be equal to TR and can be calculated as 8.5 ms as above.

Based on the above calculation, the DAR overhead for the entire Virtex-5 FPGA (say, OF_{DAR}) can be calculated as Selective Read-back (1 ms) + AF calculation and selection of recovery action (8.5 ms) + Scrubbing (8.5 ms). Hence, OF_{DAR} becomes 18 ms. As the FPGA under consideration is partially reconfigurable and contains M PRRs, the DAR overhead for each PRR (O_{DAR}) can thus be calculated as

$$O_{\text{DAR}} = \frac{\text{OF}_{\text{DAR}}}{M}. \tag{16}$$

Hence, if we partition our Virtex-5 FPGA into four equipartitioned PRRs then O_{DAR} becomes

$$O_{\text{DAR}} = \frac{\text{OF}_{\text{DAR}}}{M} = \frac{18}{4} \text{ ms} \approx 5 \text{ ms}.$$

VII. Experimental Setup

Performance evaluation of the proposed algorithm RASA has been carried out through a comprehensive set of simulation-based experiments considering a periodic real-time

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

task system where the fault occurs at a specific rate. Task success ratio (TSR) and reliability Cost (Ω_{cost}) are the principal metrics based on which the evaluation has been performed. The Ω_{cost} is defined in (14). TSR can be defined as the percentage of the total number of tasks accepted by the system over the entire schedule length out of the total number of appeared tasks. Mathematically, TSR can be formulated as

$$\text{TSR} = \frac{\nu}{\chi} \times 100\% \tag{17}$$

where ν and χ denote the total number of tasks accepted and appeared, respectively. It can be inferred that the reliability cost contributes to derive a measure of system reliability with respect to a particular schedule. On the other hand, TSR reflects how many tasks can be successfully scheduled and thus, helps to exhibit the effectiveness of the algorithm.

1) *Tasks Temporal Parameters:* Each dataset consists of randomly generated hypothetical tasks obtained from distinct distribution. In order to make, our simulation realistic, we have considered the example task sets given in [35]. Here, the authors showed that the execution time and deadline of "time-constrained" robotic tasks could vary from [100, 400], [700, 1200] time units, respectively. Hence, without loss of generality, the weights ($\text{wt}_i = [e_i/d_i]$) of the tasks have been taken from normal distribution with standard deviation $\sigma_{\text{wt}} = 0.1$ and two different values of mean, $\mu_{\text{wt}} = 0.1$, $\mu_{\text{wt}} = 0.2$. In the same way, tasks deadline have also been generated from a normal distribution with standard deviation $\sigma_d = 100$ and mean $\mu_d = 800$. Given the tasks weights, we can obtain the total workload of the system (Sys_{WL}) by summing up the weights of all the tasks. Given the system workload, the total system utilization (Sys_{uti}) can be derived by

$$\text{Sys}_{\text{uti}} = \frac{\text{Sys}_{\text{WL}}}{M} \times 100\% \tag{18}$$

where M denotes the number of PRRs. It may be noted that for a given the system utilization (Sys_{uti}), the average number of tasks (ρ) in the system can be achieved as

$$\rho = \frac{\text{Sys}_{\text{uti}} \times M}{100 \times \mu_{\text{wt}}}. \tag{19}$$

2) *Number of Instructions:* We have chosen ITC'99 Benchmark [36] task sets, in this benchmark, an arbitrary task usually contains 100 to 500 instructions for executions and in some cases even more. Without of loss of generality, we have considered that the number of instructions of our task set also vary following the normal distribution with mean 500 and standard deviation = 200.
3) *Task Spatial Parameters:* Ghorbel *et al.* [4] have demonstrated an FPGA-based implementation of robotic motion technique. They have used Virtex-5 (ML507) as their FPGA device (we have also considered the same family of FPGA) and imported different components ("edge detection," "thresholding," and "circle Hough transform") of their method at three reconfigurable regions. The size of each region was 50×38. In our simulation experiment, we also considered PRRs of similar size.

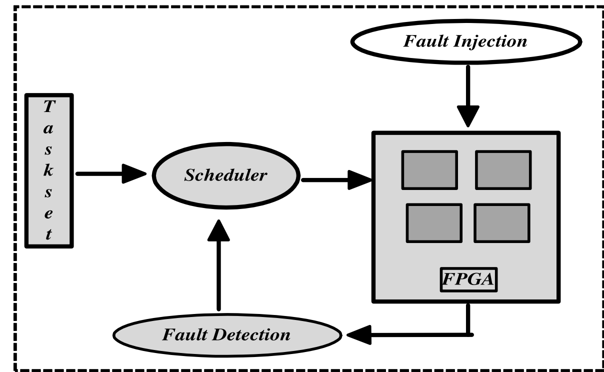

Fig. 5. Experimental setup.

After considering real-life parameters (in order to make our simulation studies more fruitful and realistic), we have generated various types of datasets by setting different values for the following parameters.

1) *Average Individual Task Weight, μ_{wt}:* The average individual task weight is given by the mean of the distribution from which task weights have been generated. Two values of μ_{wt}, 0.1 and 0.2 have been considered.
2) *System Utilization, Sys_{uti}:* We have varied the system utilization Sys_{uti} value from 40% to 90%.
3) *Fault Rate, λ:* Jacobs *et al.* [10] have depicted the average fault occurrence rate in FPGAs can vary up to 0.01/time unit. Thus, without loss of generality, we have also varied the fault rate from 0.005 to 0.04
4) *Value of Constant, K:* We have conducted our experiment by considering three distinct values of K, i.e., $K = 3, 5,$ and 10. The displayed results considered the value $K = 5$, if not stated otherwise.
5) *Number of PRRs, M:* We consider that the floor of the FPGA (Virtex-5, ML-507, XC5VFX70T) is equipartitioned into four and eight PRRs.
6) *Overheads:* As discussed in Section VI, O_{DAR} is considered as 5 and 3 ms for a 4-tiled and 8-tiled system, respectively. Full reconfiguration overhead is considered as 18 ms.

From the implementation point of view, our entire experimental setup is shown in Fig. 5. The scheduler generates the schedule and starts executing tasks as per the generated schedule. In our experimental setup, the number of PRRs remains 4 and 8. As the floor size of our considered FPGA is 38×160, then the size of each PRR will be 38×40 and 38×20, if $M = 4$ and $M = 8$, respectively. It has to be noted as included in Section II-A, each PRR's height is the integral multiple of "frame" height, i.e., 20 CLBs. Each PRR is capable of accommodating a single task at a time instant and can be dynamically reconfigured by allocating new tasks at runtime.

The "fault injection" module runs in parallel with the scheduler and selects partitions (PRRs) randomly. In a selected PRR, it injects faults (by reducing tasks execution state) with Poisson distribution at a specified fault rate. It has to be noted that at this stage, affected PRR becomes faulty and tasks

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

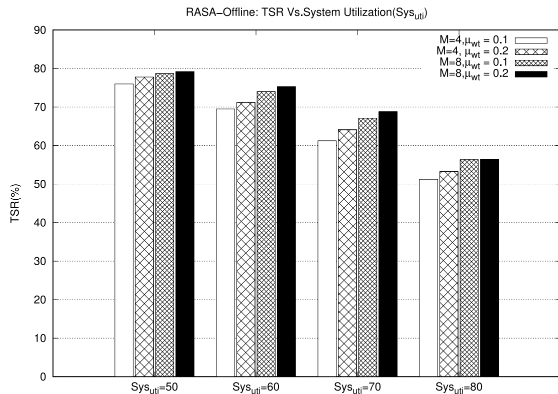

Fig. 6. TSR versus Sys_{uti}; $M = 4$ and 8.

mapped on the PRR will halt its execution. The "fault detection" component will also run in parallel with the scheduler in order to detect faults at specified "DAR" instant. Once a fault is detected it will execute the remedial action by sending commands to the scheduler.

All results are generated by running 40 different instances of each dataset type and then taking the average over these 40 distinct runs. The total schedule length is 100 000 time slots.

VIII. Results and Analysis

The performance evaluation of RASA has been carried out by measuring the task success ratio (TSR) and reliability cost (Ω_{cost}) on the different types of datasets discussed above. We will now discuss our experimental observations based on these metrics in subsequent sections.

A. Evaluation of RASA-Offline

In order to evaluate the performance of the RASA-offline, no faults have been injected in the system.

1) Task Success Ratio Versus System Utilization: Fig. 6 depicts the TSR achieved by RASA on a 4-tile and 8-tile FPGA, respectively ($M = 4$ and 8) for the varying values of system utilization (Sys_{uti}). Three insightful observations can be derived from this figure. First, as the system utilization increases the number of tasks in the system also increases [see (19), as μ_{wt} fixed] and this eventually contributes to low tasks success rate. Second, for a given system utilization, when the number of PRRs increases from $M = 4$ to $M = 8$, this makes the detection overhead low [O_{DAR}, refer (16)] and this helps to achieve higher TSR values. Third, for a fixed PRR and system utilization, if the average individual task weight (μ_{wt}) varies from 0.1 to 0.2, the achieved TSR values remain comparable. This phenomena exhibits the robustness of the proposed strategy irrespective of task's weight. In the next section, we discussed the performance comparison between the offline and online phases.

B. Evaluation of RASA-Online

In this section, we will evaluate the performance of RASA-Online. In the scheduler, faults are injected at a specified rate and the performance of the RASA-Online are studied.

TABLE I
TSR (%) Versus Fault Rate (λ) ($\text{Sys}_{\text{uti}} = 70\%$)

λ	M=4		M=8	
	μ_{wt}=0.1	μ_{wt}=0.2	μ_{wt}=0.1	μ_{wt}=0.2
0.005	65	63	71	72
0.01	56	58	64	68
0.02	44	40	55	58
0.03	37	39	47	50
0.04	31	33	42	44

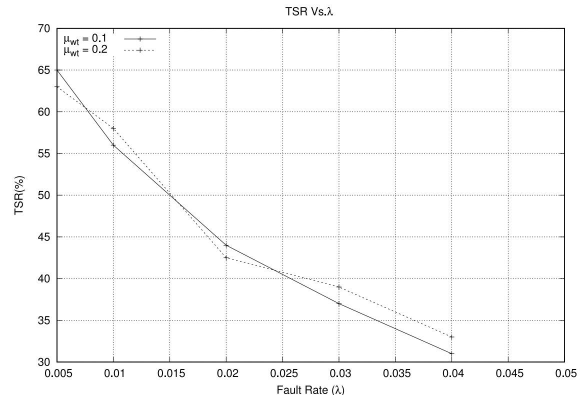

Fig. 7. TSR versus λ; $M = 4$.

1) Task Success Ratio Versus Fault Rate: Table I and Fig. 7 depict the TSR achieved by RASA on a 4-tile and 8-tile FPGA, respectively ($M = 4$ and 8) for the varying values of fault occurrence rate (λ). The average individual task weight remains 0.1 and 0.2, respectively, and the total system utilization [derived from (18)] is fixed at 70%. From the depicted table and figure, it may be observed that for a fixed Sys_{uti}, TSR decreases with increase in the fault occurrence rate (λ), with all other parameters remaining the same. This is because higher values of λ result in correspondingly more numbers of task to be affected within the hyperperiod and thus decreasing the possibility of achieving sufficient free slots by RASA-Online for remedial action in the different time windows within the hyperperiod. Insufficient free slots decrease the probability of obtaining feasible schedules for the existing tasks.

The plots in Fig. 7 show that the TSR achieved by RASA for task sets with average individual weight $\mu_{\text{wt}} = 0.1$ is comparable to that for task sets having $\mu_{\text{wt}} = 0.2$. It may be observed that for a given Sys_{uti}, the average number of tasks in sets [ρ, refer (19)] with $\mu_{\text{wt}} = 0.1$ will be nearly double compared to those having $\mu_{\text{wt}} = 0.2$. This result therefore indicates the fact that the system performs robustly against variations in the number of tasks provided at different fault occurrences as long as the Sys_{uti} remains fixed.

Another interesting observation from Table I is that the TSR increases as the number of PRRs increase from $M = 4$ to $M = 8$ when all other parameters remain the same. This is mainly due to the fact that the DAR overhead [O_{DAR}; refer (16)] decreases with an increase in the number of PRRs.

Thus, it can be concluded that with an 8-tiled FPGA system RASA can successfully tackle up to 72% of the appeared tasks by overcoming the faults (occurred at the rate 0.005) even with 70% system load.

2) Task Success Ratio Versus System Utilization: In Table II and Fig. 8, TSR is seen to decrease with an increase in system

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

TABLE II
TSR (%) VERSUS Sys_{uti} ($\mu_{\text{wt}} = 0.1$)

Sys_{uti}	M=4		M=8	
	λ=0.01	λ=0.03	λ=0.01	λ=0.03
40	81	70	88	77
50	72	62	76	69
60	66	51	71	63
70	56	37	64	47
80	47	29	53	36
90	34	22	46	29

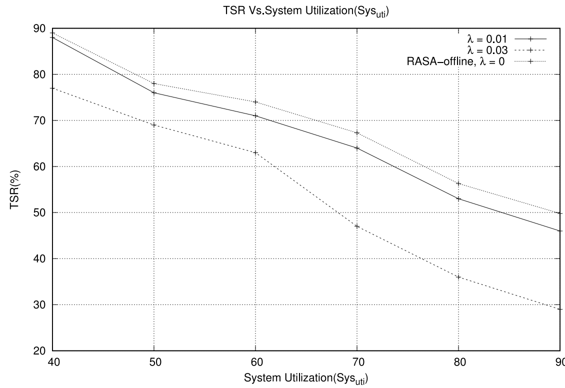

Fig. 8. TSR versus Sys_{uti}; $M = 8$.

utilization Sys_{uti}, for a given fault occurrence rate. This is because higher values of Sys_{uti} result in a correspondingly larger number of tasks (ρ), resulting in the LHS ($\sum_{j=1}^{\rho} \text{Qu}_j^{\eta}$) of (3) to become larger. Due to this, the probability of failure of the condition (3) increases for a given fault rate.

It is evident from the above results that for the same value of Sys_{uti}, when λ increases from 0.01 to 0.03, RASA suffered more task rejections and RASA with $M = 8$ produces better results compared to RASA with $M = 4$ in almost all cases, due to lower overhead (O_{DAR}). It can be observed that if the system is loaded with moderate task pressure ($\text{Sys}_{\text{uti}} = 50\%$) then RASA can successfully handle 76% tasks with a fault rate 0.01.

3) RASA-Offline Versus RASA-Online: From Fig. 8, a conclusion can be drawn with respect to performance deviation between RASA-Offline and RASA-Online, in the presence of the fault. It can be observed that with low fault occurrence rate ($\lambda = 0.01$), the achieved TSR between both the strategies is comparable. However, with high fault occurrence rate, RASA-Online still maintains the reliability of the system by reducing 10% TSR at 60% system utilization.

4) Affect of Different K Values: Fig. 9 shows the TSR obtained by RASA over varying Sys_{uti} with different values of K. We have conducted our experiments by choosing a different range of values of K and find that $K = 5$ exhibits satisfactory results in our experimental setup. In Fig. 9, the performance of RASA on achieving TSR under varying Sys_{uti} can be found with different values of K.

An interesting observation is that when the value of K becomes as small as 3 or as large as 10 then they both impose an adverse effect on the achieved TSR. This can be attributed to the fact that for lower value of K, the probability of the satisfiability of (5) decreases and thus, a late detection of fault might contribute to a lesser chance of recovery. On contrary, if the K consumes a larger value as 10 then it becomes highly

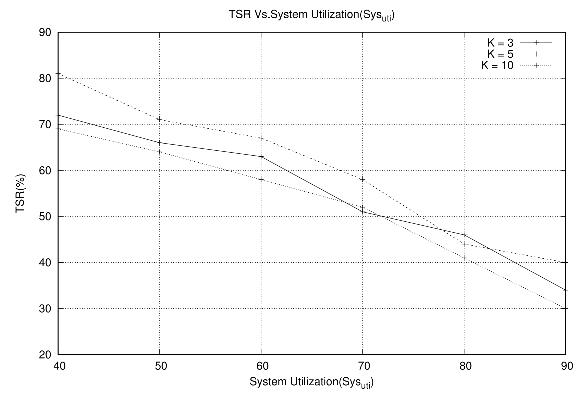

Fig. 9. TSR versus Sys_{uti}; $M = 4$; and $\mu_{\text{wt}} = 0.1$.

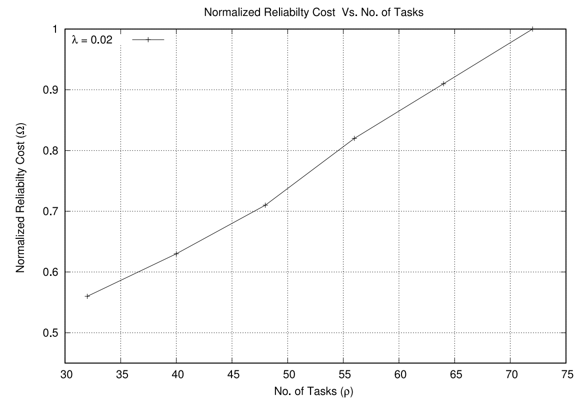

Fig. 10. $\hat{\Omega}_{\text{cost}}$ versus Sys_{uti}; $M = 8$; $\mu_{\text{wt}} = 0.1$; and $\lambda = 0.02$.

probable that (5) becomes satisfied and large number of DAR events exercise the adverse effect on reducing the "available slack" and lower the chance of recovery.

Clearly, there is a tradeoff between small or large K values. Thus, optimal values of K will be platform specific (depending upon O_{DAR}, task's deadline, length of time window).

5) Reliability Cost Versus Number of Tasks: Using (14), we have calculated the reliability cost. However, for better clarity in observing the trend of the result, we have shown the variation between normalized reliability cost (denoted as $\hat{\Omega}_{\text{cost}}$) and number of tasks ($\rho$). After normalising the value of reliability cost remains within [0, 1] and each value is obtained by dividing Ω_{cost} by $\max\{\Omega_{\text{cost}}\}$. Fig. 10 depicts that $\hat{\Omega}_{\text{cost}}$ for a certain hyperperiod increases with the increase in ρ while the number of PRRs and the fault rate remain fixed. This trend of result can be evident from (14) where the increased number of tasks will contribute to its higher value.

This can also be supported by the trend of the results obtained in the previous section. As we have already observed that the increase in the number of tasks decreases the success ratio of tasks (see Table II). Thus, the system will become less reliable and the corresponding reliability cost will be higher.

6) Comparison With Existing Works: Fig. 11 compares the performance of RASA with an existing EDF-based fault-tolerant scheduling as per [27]. Originally, this work considers much lower fault rate ("one transient fault per core per hyperperiod at random time"), low DAR overhead. It also considered fixed number of tasks (eight tasks) for all set of simulations.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

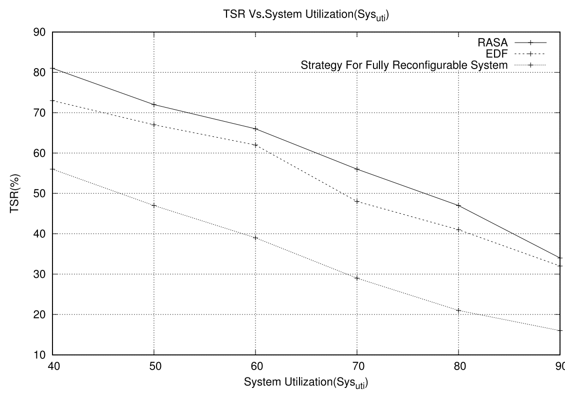

Fig. 11. TSR versus Sys_{uti}; $M = 4$; RASA versus existing technique [27].

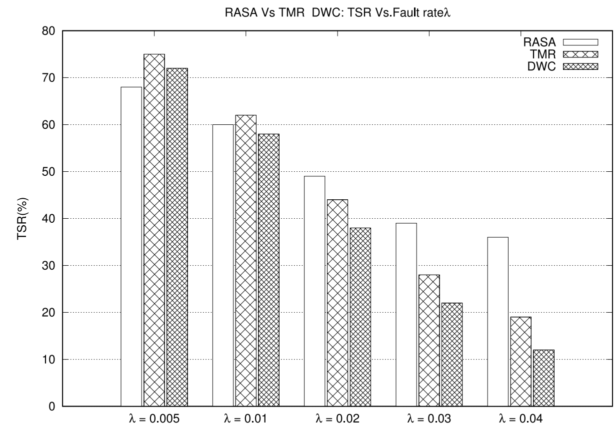

Fig. 12. RASA versus TMR and DWC.

However, for a fair comparison we have considered fault rate (λ) as 0.01, DAR overhead as 5 ms and number of tasks were varied with varying utilization [refer (19)].

It is evident from Fig. 10 that RASA comprehensively outperforms the EDF-based scrubbing scheduling technique. For example, at $\text{Sys}_{\text{uti}} = 60\%$, while RASA can successfully schedule nearly 70% of arrived tasks, the EDF-based technique can only manage approximately 50% of tasks. This is due to RASA maintaining proportional fairness, while scheduling the task set by executing them for a certain workload quota within a time window and thus, efficiently use all the PRRs. Moreover within the time window, it maintains the bitstream scrubbing (by avoiding ICAP conflicts) as a fault-tolerant method among all PRRs at a certain instant. This essentially contributes to achieving high resource utilization for RASA. On the other hand, being fixed priority-based scheme, EDF selects tasks with the closest deadlines for execution. However, if a task finishes its execution, then until its next appearance, EDF may leave some PRRs empty. However, these PRRs might be used for execution of tasks that needed to be executed earlier after recovery. Hence, EDF exhibits low resource utilization.

Fig. 11 exhibits further comparison result of the proposed RASA with strategy [13] designed for fully reconfigurable platforms. We have simulated the approach in our current experimental scenario (considering full reconfiguration overhead as 18 ms). It may be observed that, due to much lower DAR overheads ($O_{\text{DAR}} = 5$ ms), and the ability to asynchronously reconfigure individual PRRs, RASA is able to achieve much higher TSR as compared to the technique proposed in [13].

RASA has also been compared with the fault scheduling approaches as mentioned in [10]. For this comparison, we kept the number of PRRs as six ($M = 6$) to enable flexibility in placing TMR and DWC tasks. The effect on TSR values with respect to various fault rates is shown in Fig. 12. It can be observed that at a lower fault rate, the performance of TMR and DWC is comparable to RASA, however as the fault rate increases, RASA appears to maintain more reliability.

This observation can be attributed to the fact that for both TMR and DWC approaches, tasks are non preemptive in nature (hence, cannot migrate between PRRs) and are allocated into a particular PRR, until the completion of their execution time. Faults in tasks are detected by comparing or voting on the output of the replica of each task. The faulty tasks are rescheduled by executing them from the beginning. Thus at higher fault rate, when more number of tasks are affected, there are not sufficient resources for handling the tasks. On the other hand, being a preemptive scheduling strategy and with a unique recovery strategy, that does not re-execute the entire task, but executes only the affected portions via scrubbing, RASA can achieve higher success rate.

C. RASA-Online Overhead Analysis

From Algorithm 3, it can be observed that for each time boundary, almost the entire part of RASA-Online runtime overhead may be considered to be consumed at time window boundary and during DAR interval. Now, we will analyze the overheads at these intervals, individually.

1) *At Time Window Boundary:* At each time window boundary, RASA-Online starts tasks execution as per the generated offline schedule. As RASA-Online runs on separate EP (refer Section III-A), this event takes on average 400 μs. Now, it may be noted that the FPGA is bound to incur mandatory full reconfiguration overheads of the order of 18 ms (refer Section VII) at every time window boundary. The scheduling overhead of less than half a millisecond, i.e., ≈ 400 μs may be considered to be negligible with respect to the mandatory full reconfiguration overhead.
2) *During DAR Interval:* Within a time boundary, at each DAR interval, RASA-Online calculates "AF" and selects an appropriate recovery action. This event on average consumes ≈ 600 μs. However, the worst case overhead for this event is already included in the DAR overhead calculation (refer Section VI) and it should also be noted that the FPGA is bound to incur mandatory overheads (O_{DAR}) of the order of 5 ms at every DAR interval.

In both cases, RASA-Online runs on the external EP at the same time when the FPGA undergoes through a full reconfiguration at a time window boundary and a partial reconfiguration during the DAR interval. Thus, RASA-Online utilizes the mandatory preemption events (i.e., while the FPGA reconfigures) within a time boundary and does not impose any additional runtime overhead in the scheduling.

IX. Conclusion

In this work, we presented a methodology for the reliable scheduling of periodic hard real-time tasks on partially

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

reconfigurable systems deployed in EEs. Using a combined offline–online approach, the proposed methodology, called RASA, provides scheduling mechanisms with an objective of maximizing resource utilization, while keeping intermediate preemption events (checkpoints) within acceptable limits, thus favoring online detection. At design time, our strategy generates "time window" wise preemptive schedule for the arrived tasks. We have utilized these preemption points at runtime, to detect and recover any fault that occurs during execution. Simulation-based experimental results reveal that RASA is able to achieve a high task acceptance rate even under high system workloads in various fault scenarios.

References

[1] J. Jin, S. Lee, B. Jeon, T. T. Nguyen, and J. W. Jeon, "Real-time multiple object centroid tracking for gesture recognition based on FPGA," in *Proc. 7th Int. Conf. Ubiquitous Inf. Manag. Commun.*, 2013, p. 80.

[2] S. Bhasin, S. Guilley, A. Heuser, and J.-L. Danger, "From cryptography to hardware: Analyzing and protecting embedded Xilinx BRAM for cryptographic applications," *J. Cryptogr. Eng.*, vol. 3, no. 4, pp. 213–225, 2013.

[3] J. J. Rodríguez-Andina, M. D. Valdes-Peña, and M. J. Moure, "Advanced features and industrial applications of FPGAs—A review," *IEEE Trans. Ind. Informat.*, vol. 11, no. 4, pp. 853–864, Aug. 2015.

[4] A. Ghorbel, M. Jallouli, N. B. Amor, and L. Amouri, "An FPGA based platform for real time robot localization," in *Proc. Int. Conf. Individual Collective Behav. Robot. (ICBR)*, 2013, pp. 56–61.

[5] N. Gobillot, F. Guet, D. Doose, C. Grand, C. Lesire, and L. Santinelli, "Measurement-based real-time analysis of robotic software architectures," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, 2016, pp. 3306–3311.

[6] L. Jin and S. Li, "Distributed task allocation of multiple robots: A control perspective," *IEEE Trans. Syst., Man, Cybern., Syst.*, vol. 48, no. 5, pp. 693–701, May 2018.

[7] P. Maillard, J. Arver, C. Smith, O. Ballan, M. J. Hart, and Y. P. Chen, "Test methodology & neutron characterization of Xilinx 16 nm Zynq® UltraScale+TM multi-processor system-on-chip (MPSoC)," in *Proc. IEEE Nuclear & Space Radiat. Effects Conf. (NSREC)*, 2018, pp. 1–4.

[8] M. Wirthlin, "High-reliability FPGA-based systems: Space, high-energy physics, and beyond," *Proc. IEEE*, vol. 103, no. 3, pp. 379–389, Mar. 2015.

[9] R. Santos, S. Venkataraman, and A. Kumar, "Scrubbing mechanism for heterogeneous applications in reconfigurable devices," *ACM Trans. Design Autom. Electron. Syst. (TODAES)*, vol. 22, no. 2, p. 33, 2017.

[10] A. Jacobs, N. Wulf, and A. D. George, "Task scheduling for reconfigurable systems in dynamic fault-rate environments," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, 2013, pp. 1–6.

[11] F. Siegle, T. Vladimirova, J. Ilstad, and O. Emam, "Mitigation of radiation effects in SRAM-based FPGAs for space applications," *ACM Comput. Surveys*, vol. 47, no. 2, p. 37, Jan. 2015.

[12] A. Sari and M. Psarakis, "Scrubbing-aware placement for reliable FPGA systems," *IEEE Trans. Emerg. Topics Comput.*, vol. 8, no. 3, pp. 564–576, Jul.–Sep. 2020.

[13] S. Saha, S. Ehsan, A. Stoica, R. Stolkin, and K. D. McDonald-Maier, "Real-time application processing for FPGA-based resilient embedded systems in harsh environments," in *Proc. NASA/ESA Conf. Adapt. Hardw. Syst. (AHS)*, 2018, pp. 299–304.

[14] A. Gammoudi, A. Benzina, M. Khalgui, and D. Chillet, "Energy-efficient scheduling of real-time tasks in reconfigurable homogeneous multicore platforms," *IEEE Trans. Syst., Man, Cybern., Syst.*, vol. 50, no. 12, pp. 5092–5105, Dec. 2020.

[15] X. Wang, Z. Li, and W. M. Wonham, "Optimal priority-free conditionally-preemptive real-time scheduling of periodic tasks based on DES supervisory control," *IEEE Trans. Syst., Man, Cybern., Syst.*, vol. 47, no. 7, pp. 1082–1098, Jul. 2017.

[16] P. Eles, V. Izosimov, P. Pop, and Z. Peng, "Synthesis of fault-tolerant embedded systems," in *Proc. Conf. Design Autom. Test Europe*, 2008, pp. 1117–1122.

[17] A. Ejlali, B. M. Al-Hashimi, and P. Eles, "A standby-sparing technique with low energy-overhead for fault-tolerant hard real-time systems," in *Proc. 7th IEEE/ACM Int. Conf. Hardw. Softw. Codesign Syst. Synth.*, 2009, pp. 193–202.

[18] P. Pop, V. Izosimov, P. Eles, and Z. Peng, "Design optimization of time-and cost-constrained fault-tolerant embedded systems with checkpointing and replication," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 17, no. 3, pp. 389–402, Mar. 2009.

[19] Q.-H. Khuat and D. Chillet, "Communication cost reduction for hardware tasks placed on homogeneous reconfigurable resource," in *Proc. Conf. Design Archit. Signal Image Process. (DASIP)*, 2013, pp. 265–270.

[20] W. Lakhdhar, R. Mzid, M. Khalgui, Z. Li, G. Frey, and A. Al-Ahmari, "Multiobjective optimization approach for a portable development of reconfigurable real-time systems: From specification to implementation," *IEEE Trans. Syst., Man, Cybern., Syst.*, vol. 49, no. 3, pp. 623–637, Mar. 2019.

[21] T.-Y. Lee, N.-Y. Lin, W.-C. Chen, and H. Wu, "An efficient task placement method for reconfigurable FPGA systems," in *Proc. 7th Int. Conf. Complex Intell. Softw. Intensive Syst. (CISIS)*, 2013, pp. 451–455.

[22] T. Marconi, "Online scheduling and placement of hardware tasks with multiple variants on dynamically reconfigurable field-programmable gate arrays," *Comput. Electr. Eng.*, vol. 40, no. 4, pp. 1215–1237, 2014.

[23] H. Walder and M. Platzner, "Online scheduling for block-partitioned reconfigurable devices," in *Proc. Conf. Design Autom. Test Europe*, vol. 1, 2003, Art. no. 10290.

[24] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surveys*, vol. 43, no. 4, p. 35, 2011.

[25] M. Happe, A. Traber, and A. Keller, "Preemptive hardware multitasking in ReconOS," in *Proc. Int. Symp. Appl. Reconfig. Comput.*, 2015, pp. 79–90.

[26] A. Sari, M. Psarakis, and D. Gizopoulos, "Combining checkpointing and scrubbing in FPGA-based real-time systems," in *Proc. IEEE 31st VLSI Test Symp. (VTS)*, 2013, pp. 1–6.

[27] M. Psarakis and A. Sari, "A scrubbing scheduling approach for reliable FPGA multicore processors with real-time constraints," in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Nanotechnol. Syst. (DFT)*, 2017, pp. 1–4.

[28] M. H. Mottaghi and H. R. Zarandi, "DFTS: A dynamic fault-tolerant scheduling for real-time tasks in multicore processors," *Microprocess. Microsyst.*, vol. 38, no. 1, pp. 88–97, 2014.

[29] M. A. Haque, H. Aydin, and D. Zhu, "Real-time scheduling under fault bursts with multiple recovery strategy," in *Proc. IEEE 20th Real-Time Embedded Technol. Appl. Symp. (RTAS)*, 2014, pp. 63–74.

[30] A. Vega, P. Bose, and A. Buyuktosunoglu, *Rugged Embedded Systems: Computing in Harsh Environments*. Cambridge, MA, USA: Morgan Kaufmann, 2016.

[31] A. Burns, R. I. Davis, S. Baruah, and I. Bate, "Robust mixed-criticality systems," *IEEE Trans. Comput.*, vol. 67, no. 10, pp. 1478–1491, Oct. 2018.

[32] Xilinx. (2017). *Virtex-5 FPGA Configuration User Guide*. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug191.pdf

[33] K. Jozwik, H. Tomiyama, M. Edahiro, S. Honda, and H. Takada, "Comparison of preemption schemes for partially reconfigurable FPGAs," *IEEE Embedded Syst. Lett.*, vol. 4, no. 2, pp. 45–48, Jun. 2012.

[34] Xilinx. (Dec. 2014). *Virtex FPGA Data Sheet*. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds202.pdf

[35] Y. Khaluf, "Task allocation in robot swarms for time-constrained tasks," Ph.D. dissertation, Dept. Design Distribut. Embedded Syst., Univ. Paderborn, Paderborn, Germany, 2014.

[36] S. Davidson, "ITC'99 benchmark circuits-preliminary results," in *Proc. Int. Test Conf.*, 1999, pp. 1125–1125.

Sangeet Saha received the B.Tech. degree in information technology, the M.Tech. degree in computer science and engineering, and the Ph.D. degree in information technology from the University of Calcutta, Kolkata, India, in 2011, 2013, and 2018, respectively.

In 2018, he was a Research Fellow with Tata Consultancy Services, Kolkata. He has been appointed as a Senior Research Officer with the Embedded and Intelligent Systems Research Group, University of Essex, Colchester, U.K., since May 2018. His current research interests include real-time scheduling, scheduling for reconfigurable computers, real-time and fault-tolerant embedded systems, and cloud computing. He published several of his research contributions in conferences, such as CODES+ISSS, ISCAS, and NASA AHS and in journals, such as *ACM Transactions on Design Automation of Electronic Systems*, IEEE TRANSACTIONS ON MULTI-SCALE COMPUTING SYSTEMS, *Journal of Supercomputing* (Springer), and IEEE SENSORS JOURNAL.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Xiaojun Zhai (Senior Member, IEEE) received the Ph.D. degree in electrical and electronic engineering from the University of Hertfordshire, Hatfield, U.K., in 2013.

He is currently a Lecturer with the Embedded Intelligent Systems Laboratory, University of Essex, Colchester, U.K. He has authored/coauthored over 80 scientific articles in international journals and conference proceedings. His research interests include the design and implementation of the digital image and signal processing algorithms, custom computing using FPGAs, embedded systems, and hardware/software co-design.

Dr. Zhai is a member of BCS and a Fellow of HEA.

Shoaib Ehsan (Senior Member, IEEE) received the B.Sc. degree in electrical engineering from the University of Engineering and Technology, Taxila, Pakistan, in 2003, and the Ph.D. degree in computing and electronic systems (with specialization in computer vision) from the University of Essex, Colchester, U.K., in 2012.

He has an extensive industrial and academic experience in the areas of embedded systems, embedded software design, computer vision, and image processing. His current research interests are in intrusion detection for embedded systems, local feature detection and description techniques, and image feature matching and performance analysis of vision systems.

Dr. Ehsan was a recipient of the University of Essex Post Graduate Research Scholarship, the Overseas Research Student Scholarship, and the prestigious Sullivan Doctoral Thesis Prize awarded annually by the British Machine Vision Association.

Shakaiba Majeed received the M.S. degree in electronics engineering with thesis on interval type-2 fuzzy systems and the Ph.D. degree in computer and software engineering contributing additional knowledge in testing and debugging of event-based software from Hanyang University, Seoul, South Korea, in 2010 and 2018, respectively.

Her current research interests include real-time embedded software design and analysis, real-time operating systems/middleware, and open-source software.

Klaus McDonald-Maier (Senior Member, IEEE) received the Dipl.-Ing. and M.S. degrees in electrical engineering from the University of Ulm, Ulm, Germany, and CPE-Lyon, Villeurbanne, France, in 1995, and the Doctorate degree in computer science from the Friedrich-Schiller-University, Jena, Germany, in 1999.

He is currently the Head of the Embedded and Intelligent Systems Laboratory, University of Essex, Colchester, U.K. He is also the Chief Scientist with UltraSoC Technologies Ltd., Cambridge, U.K., the CEO of Metrarc Ltd., Cambridge, and a Visiting Professor with the University of Kent, Canterbury, U.K. His current research interests include embedded systems and system-on-chip design, security, development support and technology, parallel and energy-efficient architectures, computer vision, data analytics, and the application of soft computing and image processing techniques for real-world problems.

Dr. McDonald-Maier is a member of VDE and a Fellow of BCS and IET.