A Framework for Supporting Real-Time Applications on Dynamic Reconfigurable FPGAs

Alessandro Biondi, Alessio Balsini, Marco Pagani, Enrico Rossi, Mauro Marinoni, Giorgio Buttazzo
Scuola Superiore Sant'Anna, Pisa, Italy
Email: {alessandro.biondi, alessio.balsini, marco.pagani, enrico.rossi, mauro.marinoni, giorgio.buttazzo}@sssup.it

***Abstract*—Computing platforms are evolving towards heterogeneous architectures including processors of different types and field programmable gate arrays (FPGAs), used as hardware accelerators for speeding up specific functions. The increasing capacity and performance of modern FPGAs, with their partial reconfiguration capabilities, have made them attractive in several application domains, including space applications.**

This paper proposes a framework for supporting the development of safety-critical real-time systems that exploit hardware accelerators developed through FPGAs with dynamic partial reconfiguration capabilities.

A model is first presented and then used to derive a response-time analysis to verify the schedulability of a real-time task set under given constraints and assumptions. Although the analysis is based on a generic model, the proposed framework has been conceived to account for several real-world constraints present on today's platforms and has been practically validated on the Zynq platform, showing that it can actually be supported by state-of-the-art technologies. Finally, a number of experiments are reported to evaluate the worst-case performance of the proposed approach on synthetic workload.

I. Introduction

Current computer architectures are evolving towards heterogeneous platforms consisting of hybrid computational devices that may include processors of different types and field programmable gate arrays (FPGAs). In particular, the reprogrammable and reconfigurable capabilities of FPGAs, their increasing capacity, and their suitability for signal processing have made them attractive in several application domains, as alternatives to application specific integrated circuits (ASICs) [1].

Xilinx [2] provided an analysis of recent progress in field programmable logic, highlighting that FPGAs have become bigger (comprising several million gates and up to a million bits of on-chip memory), faster (allowing system clock rates up to 200 MHz and I/O speed of up to 800 Mbits/second), more versatile (featuring dedicated carry structures to support adders, accumulators and counters), and cheaper, in terms of cost per logic gate.

Reprogrammable FPGA featuring high flexibility, combined with high performance and complexity are becoming increasingly important for space applications. With satellite lifetimes increased far beyond 10 years, re-programmability in flight becomes a stringent requirement. Moreover, in space environments, where radiation can cause bit flips in memory elements and ionisation failure in semiconductors, the use of reconfigurable hardware allows modifying on-board functions by replacing faulty/outdated designs at different stages of a mission.

Modern FPGA chips allow dynamic partial reconfiguration (DPR) capabilities, enabling the user to reconfigure a portion of the FPGA dynamically (at runtime), while the remainder of the device continues to operate [3]. This is especially valuable in mission-critical systems that cannot be disrupted while some subsystems are being redefined. In this context, mission-critical functions could continue to meet external interface requirements while other reconfiguration regions are reprogrammed to provide different functionality.

Partial reconfiguration is also useful in systems where multiple functions share the same FPGA resources. In such systems, one section of the FPGA continues to operate, while other sections are reconfigured to provide new functionality. Such an interesting capability opens a new scheduling dimension for applications running on heterogeneous platforms. As in multitasking, where multiple applications share the processors by switching contexts between software processes, DPR enables the possibility of interleaving multiple functions implemented as programmable logic on an FPGA recurrently shared by different processing components. However, this is possible at the cost of reconfiguration times, which - today - are three orders of magnitude higher than context switch times in multitasking.

Despite this limitation, there is a clear evolution trend showing that reconfiguration times are progressively decreasing. Liu et al. [4] designed a smart reconfiguration peripheral interface, based on the Xilinx ICAP port [5], that is able to approach a throughput of 400 MB/s. Also, Duhem et al. [6] designed a fast reconfiguration interface by overclocking the ICAP port up to 200 MHz, corresponding to a throughput of 800 MB/s. An overview of the trend of reconfiguration times (obtained by comparing the theoretical maximum throughput calculated from platforms' datasheets) is shown in Figure 1, based on the study conducted by Pagani et al. [7]. For this reason, it is plausible to expect that such a trend will continue in the upcoming years, thus making DPR a relevant direction to be explored.

Although reconfiguration times are not negligible, FPGAs allow hardware acceleration of a wide class of algorithms with a significant speedup factor [8], [9] over the corresponding sequential software implementation. For instance, in the case study analyzed in this work, a speedup factor up to 15x has been measured for an image processing filter implemented on the Zynq-7010 platform, which can reach a throughput of 145 MB/s for the DPR, allowing to reconfigure an FPGA area containing about 25% of the total resources in less than 3 milliseconds.

When exploiting FPGAs with DPR in real-time embedded systems, a crucial issue is to provide worst-case response time bounds of computations consisting of software tasks and hardware accelerated functions. Although several works have been done to analyze the timing behavior of real-time applications

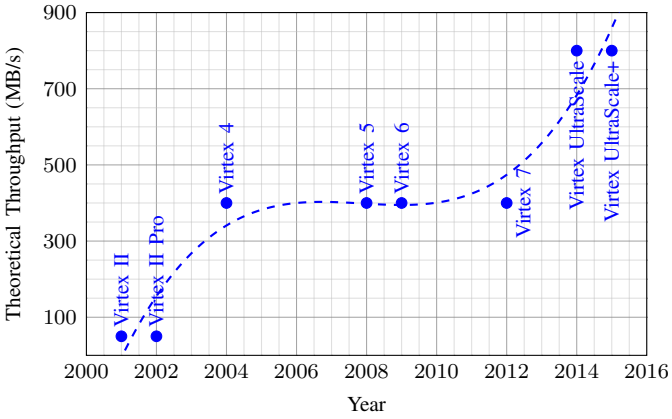

Figure 1. Trend of reconfiguration throughput.

using FPGAs, most of them did not consider DPR capabilities at a job level. To overcome this lack, this work proposes a new computing framework for enabling a timing analysis of real-time activities that make use of hardware accelerators developed through programmable FPGAs with DPR capabilities.

Contributions. This paper provides the following three main novel contributions:

1) It presents FRED, a framework for supporting real-time applications on FPGAs with DPR feature. It relies on a *static* off-line partitioning of the FPGA fabric to limit worst-case scenarios arising when using DPR. Design issues related to scheduling and inter-task communication are also discussed for bounding worst-case delays.
2) It proposes a new task model to abstract a set of real-time activities running on the considered architecture.
3) It derives a response-time analysis to verify the schedulability of a set of real-time tasks consisting of both software parts and hardware accelerated functions.

Although based on a generic modeling abstraction, the proposed framework has been conceived by considering several real-world constraints that are present on today's platforms. In fact, FRED has been also practically validated with a proof-of-concept implementation on the Zynq platform [10]. Such a practical validation highlighted that the proposed approach can be actually supported by state-of-the-art technologies with a limited run-time overhead. Finally, to explore the worst-case performance of FRED, an empirical study (based on synthetic workload) has been conducted to evaluate the proposed response-time analysis under different operating scenarios. [1]

Paper organization. The remainder of this paper is organized as follows. Section II presents a taxonomy of the different approaches and then discusses the related work. Section III describes the proposed framework and the related models. Section IV derives a response-time analysis for a set of tasks that use hardware acceleration. Section V presents some experimental results carried out on the Zynq platform to validate the proposed approach. Section VI reports a set of experiments aimed at evaluating the performance of the proposed response-time analysis on synthetic workload. Section VII concludes the paper and highlights some future work.

[1] *Artifact Evaluation* (AE) instructions are available at http://retis.sssup.it/~a.biondi/ae/FRED/. The artifact has been accepted by the RTSS AE committee.

II. Related work

The solutions proposed in the literature to exploit FPGA acceleration are quite heterogeneous due to the evolution of such platforms and the wide range of applications that can take advantage of this technology. The intrinsic parallelism, the reduced interference among the running activities, and the reduced variability in the execution made such a technology appealing for real-time applications, ranging from network management [11] to scheduling of hard [12] and soft [13] tasks. However, these solutions are limited to static or slowly evolving scenarios. Before analyzing the related work on DPR for real-time task scheduling, a taxonomy is first introduced to classify the existing solutions and precisely position the proposed approach with respect to the literature.

Taxonomy. The features considered to organize the taxonomy concern the reconfiguration approach, the allocation methods, the model of the *FPGA reconfiguration interface* (FRI), and the types of managed tasks.

Reconfiguration approaches. They can be distinguished between *static* and *dynamic*. In a static approach, the allocation of hardware tasks (HW-tasks) is performed at the initialization phase, while in a dynamic approach HW-tasks can be allocated at runtime upon specific events. Dynamic approaches can be used to support *mode-changes* in the application (allowing tasks to be added and removed from the task set) or trigger a reconfiguration every time a new job is scheduled (*job-level* reconfiguration). A static approach has no runtime reconfiguration overhead, but the maximum number of HW-tasks is limited by the physical size of the FPGA. Dynamic approaches trade extra reconfiguration overhead to increase the total number of HW-tasks that can be managed.

Allocation methods. They can be distinguished between *slotted* and *slotless*. In a slotted approach, the FPGA area is partitioned into slots of given size connected via buses provided on the static part of the FPGA. A HW-task can occupy one or more slots. In a slotless solution, HW-tasks can arbitrarily be positioned on the FPGA area and data are transferred through the reconfiguration interface inside the FPGA. Slotted approaches have the advantage of having the communication channels already in place, but the FPGA area may be partially wasted due to slot granularity. On the other hand, slotless solutions increase the utilization efficiency of the FPGA area, but are penalized by higher reconfiguration times due to the instantiation of communication channels and the increased traffic on the FRI due to the additional data transfer.

FRI model. The FRI plays a central role in FPGAs with DPR, thus, building a proper model of the FRI is crucial for estimating worst-case delays and enabling a real-time analysis. The easiest approach is to reduce complexity by considering reconfiguration delays negligible. This is a strong unrealistic assumption, considering that, in current FPGAs, reconfiguration delays can have the same order of magnitude of task execution times. A simple approximation can be obtained using a constant reconfiguration time. However, since the reconfiguration time is proportional to the number of elements to be reconfigured, and the FRI is a shared resource, providing a safe bound would introduce a huge pessimism in the analysis. Less pessimistic

values can be obtained considering the reconfiguration time composed by two elements: one proportional to the number of elements to be reconfigured and one due to the time spent in waiting for the FRI. Most of the works focused on kernel mechanisms considered an FRI model tailored to real solutions, as the Xilinx ICAP port [5].

Task model. Modern heterogeneous platforms include FPGAs modules together with processors on the same chip [14]. On such platforms it is thus possible to execute both HW-tasks, running on the FGPA, and software tasks (SW-tasks), running on the processors.

Related work analysis. The works considered in this section are related to the proposed approach in that they provide a timing analysis under reconfigurable FPGA architectures or propose a software support for HW-task management.

Di Natale and Bini [12] proposed an optimization method to partition the reconfigurable area of a homogeneous FPGA platform into slots to be allocated to HW-tasks and softcores running the remaining tasks. Given the high computational complexity of the method, this approach can only be used off-line to obtain a static task allocation, hence it does not exploit the advantages of the dynamic reconfiguration. Pellizzoni and Caccamo [15] addressed a similar problem in a more dynamic scenario, proposing an allocation scheme and an admission test to provide real-time guarantees of applications supporting mode changes, where tasks can either be executed in software on a CPU or in hardware on the FPGA.

Danne and Platzner [16] presented two algorithms (one EDF-based and one server-based) to schedule only preemptive HW-tasks, but the model adopted for the FPGA platform is quite simple and does not consider any reconfiguration time and allocation constraints. Saha et. al. [17] presented a new scheduling algorithm for preemptable HW-tasks, exploiting the higher speed and the improved capabilities of modern reconfiguration interfaces to dynamically change the allocation every time a task terminates. However, this approach assumes a homogeneous partition and a fixed reconfiguration time, which can lead to a huge waste of the area and a high pessimism in the analysis. In summary, in all the works cited above, the models used for the FPGA and the reconfiguration interface are too simple to describe the limitations of the available platforms, and the corresponding approaches do not fully exploit reconfiguration capabilities under real-time constraints.

Dittmann and Frank [18] addressed the analysis of reconfiguration requests as a single core scheduling problem. The paper assumes a single set of homogeneous slots managed by a non-preemptable FRI and considers only HW-tasks (SW-tasks are not taken into account). Unfortunately, due to missing proofs, it is not clear how response-time bounds follow. In addition, the authors did not investigate sustainability issues and their analysis may be affected by later-discovered misconceptions concerning non-preemptive fixed-priority scheduling [19].

Other authors proposed methods for supporting a job-level reconfiguration from a system perspective. The easiest solution is to communicate with a HW-task through proper software stubs that interact with the kernel scheduler and manage the HW-tasks at the application level. Another approach is to extend the operating system to provide specific primitives for scheduling, allocating, and programming HW-tasks, along with those related to SW-tasks management. For instance, Lübbers and Platzner [20] proposed the ReconOS operating system, which extends the classic multi-threading programming model to hardware activities executed on a reconfigurable device. Originally designed for fully reconfigurable FPGAs, this solution has then been extended by the same authors to support partial reconfiguration [21], with a cooperative multitasking approach to deal with slot contentions. More recently, Happe et. al. [22] proposed an extension to the ReconOS execution environment to provide HW-tasks preemptability. However, its focus is on hardware enabling technologies, not on a kernel support for exploiting this capability. Iturbe et al. [23] presented the R3TOS operating system to support a more dynamic task allocation, exploiting the reconfiguration interface to avoid preconfigured static communication channels. The authors proposed a HW-task model and algorithms for their scheduling and allocation. However, a worst-case analysis is not provided and nothing is said on the schedulability of SW-tasks.

Although based on a more realistic FPGA model, the approaches considered in this second set of papers have been designed to improve the average system performance and focused on kernel implementation issues, without deriving worst-case response times bounds. As a consequence, these methods cannot be used for a real-time scheduling analysis.

Classification. Table I classifies the presented papers according to the proposed taxonomy, also highlighting the availability of a real-time analysis (RTA) to better emphasize the differences with respect to the proposed approach. Summarizing, different approaches have been proposed to exploit the advantages of DPR-enabled FPGAs, but none of them provided worst-case bounds for enabling a worst-case timing analysis of real-time sets of mixed HW-tasks and SW-tasks. In addition, most of the previous work did not consider heterogeneous FPGA slots. To overcome these limitations, the work proposed in this paper presents a heterogeneous slotted-based framework designed to make reconfiguration times more predictable and derive a schedulability analysis for real-time applications exploiting DPR capabilities. FPGA reconfiguration is managed at the job level and the schedulability analysis takes into account the delays and the constraints coming from the FRI. Both preemptive and non-preemptive reconfiguration are analyzed.

Paper	Reconfig.	Alloc.	FRI model	Tasks	RTA
Lübbers, 09	Static	Slotted	ICAP	HW/SW	No
Lübbers, 10	Job-level NP	Slotted	ICAP	HW/SW	No
Happe, 15	Job-level P	Slotted	ICAP	HW/SW	No
Iturbe, 15	Job-level NP	Slotless	ICAP	HW/SW	No
Di Natale, 07	Static	Slotless	*Not required*	HW/SW	Yes
Pellizzoni, 07	Mode-ch NP	Slotted	*Not addressed*	HW/SW	Yes
Danne, 05	Job-level P	Slotless	*Zero overhead*	HW	Yes
Saha, 15	Job-level P	Slotless	*Fixed overhead*	HW	Yes
Dittmann, 07	Job-level NP	Slotted	General (NP)	HW	Yes
This paper	Job-level NP	Slotted	General (P/NP)	HW/SW	Yes

Table I
CLASSIFICATION OF THE RELATED WORK.

III. FRAMEWORK AND MODELING

This work considers a platform consisting of a processor and a DPR-enabled FPGA module that comprises b logic blocks.

The FPGA and the processor share a common memory $\mathcal{M}$. The blocks of the FPGA module are statically partitioned into a set $P = \{P_1, ..., P_{n_P}\}$ of n_P *partitions*, where each partition P_k is composed of b_k logic blocks, with $\sum_{k=1}^{n_P} b_k \leq b$. Blocks are not shared among partitions. Furthermore, each partition P_k is split into n_k^S *slots* of b_k^S logic blocks, such that $\forall P_k \in P,\ n_k^S \cdot b_k^S \leq b_k$. Blocks are not shared among the slots.

As described in Section II, a slotted approach is more suitable for real-time systems because reconfiguration delays are shorter and more predictable than in a slotless solution, since there is no overhead related to task allocation management and to instantiation of communication channels. On the other hand, a slotted approach introduces a granularity that may increase the wasted area of the FPGA. This phenomenon can be mitigated by a proper design of slots and partitions as a function of the tasks. However, this issue is not addressed in this paper due to space limitations.

A. Hardware task model

The activities executed on the FPGA are modeled as a set $\Gamma^H = \{\tau_1^H, ..., \tau_{n_H}^H\}$ of n_H HW-tasks. Each HW-task τ_i^H requires b_i logic blocks and has a *worst-case execution time* (WCET) C_i^H. A HW-task can execute only if it has been programmed on a slot of the FPGA.

The considered platform is equipped with a *FPGA reconfiguration interface* (FRI) able to dynamically reconfigure a slot at run-time by programming a specific HW-task τ_i^H. Each slot can accommodate at most one HW-task [5], [24]. As true in real-world platforms (such as [25], [26]), we assume that

(i) the FRI can reconfigure a slot without affecting the execution of the HW-tasks currently running in other slots;
(ii) no processor cycles are used for reconfiguring a slot (i.e., the FRI is an external peripheral, like DMA [14]); and
(iii) the FRI can program at most one slot at a time.

To program a given HW-task τ_i^H into a slot, the FRI has to program all its logic blocks, independently of the number b_i of logic blocks required by τ_i^H, because unused blocks have to be disabled to "clean" the previous slot configuration.

Each HW-task τ_i^H can be programmed in any of the slots belonging to a single partition. The partition hosting a HW-task τ_i^H is denoted as $P(\tau_i^H)$ and referred to as *affinity*. For all HW-tasks with affinity $P(\tau_i^H) = P_k$, it must be $b_i \leq b_k^S$.

The FRI is characterized by a *throughput* ρ, meaning that $r_k^S = b_k^S/\rho$ units of time are needed to program a slot of a given partition P_k. Hence, the time r_a needed to program a HW-task τ_a^H is $r_a = r_k^S \ :\ P(\tau_a^H) = P_k$.

B. Software task model

The activities executed on the processor are modeled as a set $\Gamma^S = \{\tau_1, ..., \tau_{n_S}\}$ of n_S SW-tasks. Each SW-task can make use of HW-tasks to accelerate specific functions and is subject to timing constraints. In particular, each SW-task τ_i

- uses a set $\mathcal{H}(\tau_i) \subseteq \Gamma^H$ of m_i HW-tasks;
- alternates the execution of $m_i + 1$ *sub-tasks* (also referred to as *chunks*) with the execution of the m_i HW-tasks in $\mathcal{H}(\tau_i)$; thus, the execution of a SW-task τ_i can be represented as a sequence $\tau_i := \langle \tau_{i,1}, \tau_a^H, \tau_{i,2}, \tau_b^H, \ldots, \tau_{i,m_i+1} \rangle$, where $\{\tau_a^H, \tau_b^H, \ldots\} \in \mathcal{H}(\tau_i)$ and $\tau_{i,j}$ is the j-th sub-task of τ_i. Whenever the execution of a HW-task τ_a^H is requested, the corresponding SW-task *self-suspends* until the completion of τ_a^H. The beginning of the self-suspension phase coincides with the termination of the sub-task that issued a request for a HW-task. In a dual manner, the completion of a HW-task coincides with the release of the next sub-task.
- has a total WCET C_i, composed of the WCETs $C_{i,j}$ of all its sub-tasks $\tau_{i,j}$; that is, $C_i = \sum_{j=1}^{m+1} C_{i,j}$.
- is periodically (or sporadically) released with a period (or minimum inter-arrival time) of T_i units of time, thus generating an infinite sequence of execution instances (denoted as jobs);
- is subject to timing constraints; that is, each of its jobs must complete its execution within a *deadline* D_i relative to its activation time.

Each HW-task can be used by at most one SW-task, that is $\bigcap_{\tau_i \in \Gamma^S} \mathcal{H}(\tau_i) = \emptyset$.

Figure 2 reports the pseudo-code defining the implementation skeleton of a SW-task τ_i that uses $m_i = 2$ HW-tasks in the set $\mathcal{H}(\tau_i) = \{\tau_a^H, \tau_b^H\}$. The statement <...> has been used to represent a generic set of instructions that are part of a computation executed by the SW-task on the processor.

```
1  TASK(τ_i)
2  {
3    <...>
4    <prepare input data for τ_a^H>
5    EXECUTE_HW_TASK(τ_a^H);
6    <retrieve output data from τ_a^H>
7    <...>
8    <prepare input data for τ_b^H>
9    EXECUTE_HW_TASK(τ_b^H);
10   <retrieve output data from τ_b^H>
11   <...>
12 }
```

Figure 2. Pseudo-code of the implementation skeleton of a SW-task.

The SW-task illustrated in Figure 2 is described by the sequence $\langle \tau_{i,1}, \tau_a^H, \tau_{i,2}, \tau_b^H, \tau_{i,3} \rangle$: the first sub-task $\tau_{i,1}$ consists of lines 3-5, the second sub-task $\tau_{i,2}$ of lines 6-9 and the third sub-task $\tau_{i,3}$ of lines 10-11. `EXECUTE_HW_TASK(`τ_j^H`)` is a *blocking* system call, which is in charge of (i) requesting the execution of τ_j^H and (ii) *suspending* the execution of τ_i until the completion of τ_j^H. Note that at line 4, $\tau_{i,1}$ prepares the input data for τ_a^H. Similarly, $\tau_{i,2}$ retrieves the output data produced by τ_a^H (line 6) and prepares the input data for τ_b^H (line 8). Further details on the inter-task communication mechanism are discussed in Section III-D.

Figure 3 illustrates the execution behavior of another SW-task $\tau_i := \langle \tau_{i,1}, \tau_a^H, \tau_{i,2} \rangle$, visualizing the delays experienced when requesting the execution of τ_a^H.

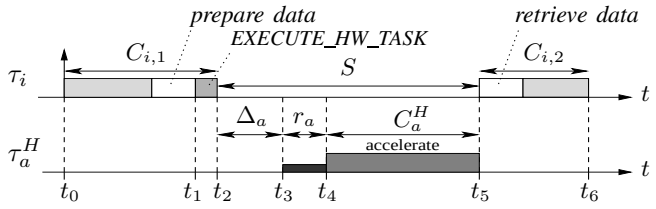

Figure 3. Execution behavior of a SW-task calling a HW-task.

As clear from the figure, task τ_i is activated at time t_0. At time t_1, the first sub-task $\tau_{i,1}$ requests the execution of the HW-task τ_a^H and self-suspends its execution at time t_2, where (t_2-t_1) corresponds to the system overhead to issue the request. This example assumes that all the slots of partition $P(\tau_a^H)$ are busy (i.e., occupied by other HW-tasks that are currently executing), hence a delay Δ_a is introduced from time t_2 until time t_3, at which one slot of $P(\tau_a^H)$ becomes free. Once there is a free slot in $P(\tau_a^H)$, the HW-task can be programmed, from time t_3 to t_4, by using the FRI: such an operation takes at most r_a units of time, where $r_a = b_k^S/\rho$ (being k the affinity of τ_a^H).

After the programming phase, τ_a^H starts executing at time t_4 on the FPGA and completes at time t_5 within C_a^H units of time. Then, the SW-task is resumed and executes the second sub-task $\tau_{i,2}$, which completes at time t_6. Note that τ_i is suspended for the interval $[t_2, t_5]$, which is no longer than $S = \Delta_a + r_a + C_a^H$.

While the example presented above has a single SW-task, the system considered in this paper includes multiple SW-tasks and HW-tasks that contend the resources available on the platform. This means that a SW-task τ_i can suffer a temporal *interference* from the execution of other SW-tasks that, if not properly managed, can determine the violation of its deadline D_i. Such interference also depends on the contention for the FPGA slots and the FRI caused by the other HW-tasks. For such reasons, a *scheduling infrastructure* is needed to support a set of concurrent HW-tasks and SW-tasks.

The symbols used in the paper are summarized in Table II.

b	total number of logic blocks in the FPGA
n_P	number of partitions in the FPGA
P_k	k-th partition in the FPGA
b_k	number of logic blocks in partition P_k
b_k^S	number of logic blocks in a slot of P_k
n_k^S	number of slots in partition P_k
ρ	throughput of the reconfiguration interface
r_k^S	time to program a slot of partition P_k
n_S	number of software tasks
n_H	number of hardware tasks
τ_i	i-th software task
$\tau_{i,j}$	j-th sub-task of the i-th software task
τ_a^H	a-th hardware task
$P(\tau_a^H)$	partition hosting the a-th hardware task
r_a	time to program the HW-task τ_a^H
C_a^H	worst-case execution time of HW-task τ_a^H
Δ_a	delay experienced by τ_a^H to wait for a free slot
b_a	number of logic blocks required by τ_a^H
C_i	worst-case execution time of SW-task τ_i
$C_{i,j}$	worst-case execution time of sub-task $\tau_{i,j}$
π_i	priority assigned to SW-task τ_i
T_i	period (or minimum inter-arrival time) of τ_i
D_i	relative deadline of SW-task τ_i
m_i	number of HW-tasks used by SW-task τ_i

Table II
SYMBOLS USED THROUGHOUT THE PAPER.

C. Scheduling infrastructure

Each SW-task τ_i is assigned a fixed priority π_i, also inherited by all its sub-tasks. A SW-task is denoted as *ready* when (i) it has a pending job (i.e., a job released but not yet completed) and (ii) it is not self-suspended waiting for the completion of a HW-task. SW-tasks are assumed to be scheduled according to a fixed-priority (FP) preemptive scheduling algorithm, so that, at any point in time, the ready task with the highest priority is executed on the processor.

Besides the processor, two other resources are contented by SW-tasks: the slots in the FPGA partition (shared with other HW-tasks having the same affinity) and the FRI. Hence, multiple requests for such resources have to be scheduled. The overall scheduling infrastructure managing the slots and the FRI is based on a multi-level queue structure, illustrated in Figure 4.

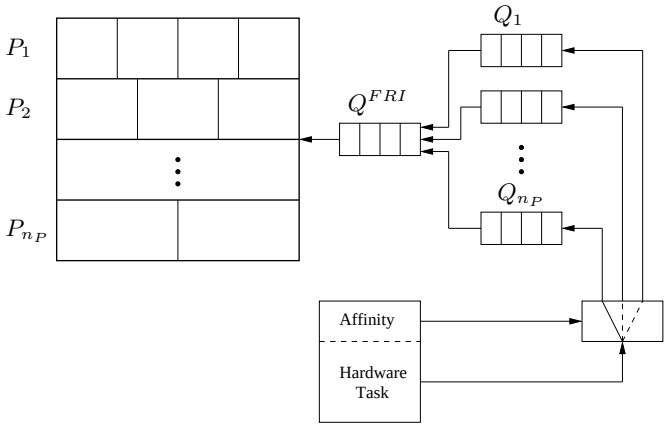

Figure 4. Scheduling infrastructure for HW-tasks requests in FRED.

We begin by describing the scheduling policies used for each resource; then, we present the scheduling rules that apply to every request for HW-tasks when traversing the multi-level queue structure of Figure 4.

Slot scheduling. For the purpose of scheduling, each slot can be *free* or *busy*. A busy slot can in turn be *active*, when there is a HW-task programmed on it that is executing, or *reserved*. A HW-task τ_i^H with affinity $P(\tau_i^H) = P_k$, that is waiting for a free slot in partition P_k, is kept in a queue Q_k managed according to a first-in-first-out (FIFO) policy. Note that such a scheduling policy guarantees a starvation-free progress mechanism. Moreover, it does not require preempting the execution of HW-tasks, which is known to be a challenging issue [27] [18] leading to non-negligible run-time overheads.

FRI scheduling. Whenever there are x free slots into a given partition P_k, such x slots are *reserved* for the first x HW-task requests waiting into Q_k which then have to contend the FRI to program their corresponding HW-task. While slots are shared only among the HW-tasks belonging to the same partition, the FRI is a single resource contented by all the requests for HW-tasks in the system. HW-task requests contending the FRI are kept in a queue denoted as Q^{FRI}.

In this paper, slot programming requests are managed according to a *ticket-based scheduling* policy, which is described below and can be configured to be executed either in a preemptive or non-preemptive fashion. Please note that HW-task execution is assumed to be non preemptive to contain the preemption overhead associated to FPGA reconfigurations. Hence, here preemptive and non-preemptive policies are only related to the FRI programming phase. Under a non-preemptive policy, the programming phase cannot be interrupted, whereas under a preemptive policy, the programming phase can be interrupted to serve another programming request.

Ticket-based scheduling. The ticked-based scheduling policy is described by the following rules that apply to both non-preemptive and preemptive management of the FRI:

R1 Each execution request $\mathcal{R}_a$ for an HW-task τ_a^H is assigned a "ticket" marked with the absolute time $t(\mathcal{R}_a)$ at which $\mathcal{R}_a$ has been issued.

R2 Every partition queue Q_k and the FRI queue Q^{FRI} enqueues execution requests for HW-tasks by *increasing* ticket time.

R3 When a request $\mathcal{R}_a$ for HW-task τ_a^H is issued, $\mathcal{R}_a$ is inserted in the partition queue Q_k with $P_k = P(\tau_a^H)$.

R4 At any point in time t, for every partition queue Q_k, the first $\eta_k(t) \geq 0$ requests in Q_k are removed from Q_k and inserted in Q^{FRI}, where $\eta_k(t)$ is the number of *free* slots in P_k at time t. Contextually, these $\eta_k(t)$ slots become *reserved* (and hence *busy*).

R5 Once the HW-task τ_a^H related to a request $\mathcal{R}_a$ has been programmed onto a slot, $\mathcal{R}_a$ is removed from Q^{FRI}, that slot becomes *active*, and τ_a^H starts executing.

R6 When a HW-task τ_a^H completes its execution, the corresponding slot becomes *free*.

The following scheduling rules distinguish between non-preemptive and preemptive management of the FRI. In the case of preemptive FRI scheduling, the following rule holds:

R-P1 Whenever Q^{FRI} is not empty, the FRI programs the HW-task related to the first request in Q^{FRI} (i.e., the one having the earliest ticket time).

For non-preemptive FRI scheduling the following rules hold:

R-NP1 When the FRI is programming a HW-task it cannot be interrupted to serve another request.

R-NP2 When the FRI completes a programming phase, or Q^{FRI} becomes not empty, the FRI starts programming the HW-task related to the first request in Q^{FRI}.

Example. Figure 5 shows an example of preemptive FRI management schedule under FRED for an FPGA module containing two partitions P_1 and P_2, each consisting of a single slot.

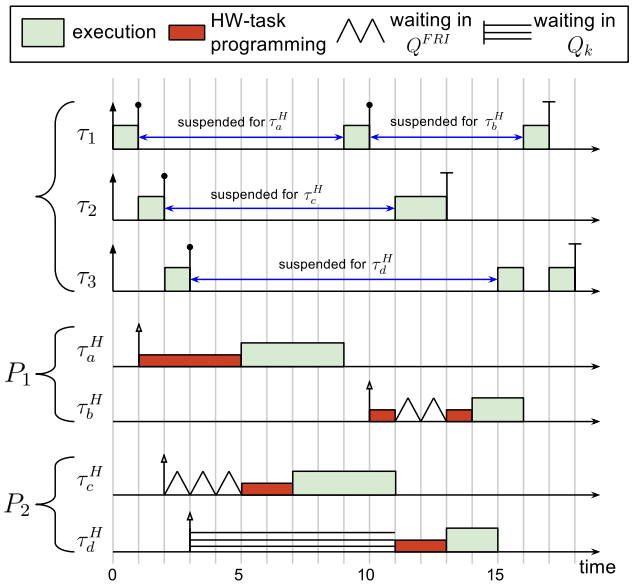

Figure 5. Example of preemptive FRI scheduling under FRED.

The application consists of three SW-tasks: $\tau_1 = \langle \tau_{1,1}, \tau_a^H, \tau_{1,2}, \tau_b^H, \tau_{1,3} \rangle$, $\tau_2 = \langle \tau_{2,1}, \tau_c^H, \tau_{2,2} \rangle$, and $\tau_3 = \langle \tau_{3,1}, \tau_d^H, \tau_{3,2} \rangle$. The priority assignment is such that $\pi_1 > \pi_2 > \pi_3$. HW-tasks τ_a^H and τ_b^H share partition P_1 (i.e., $P(\tau_a^H) = P(\tau_b^H) = P_1$), whereas HW-tasks τ_c^H and τ_d^H share partition P_2 (i.e., $P(\tau_c^H) = P(\tau_d^H) = P_2$).

All the SW-tasks are synchronously released at time 0. Being the highest-priority one, τ_1 starts executing as first and at time $t = 1$ completes its sub-task $\tau_{1,1}$ by issuing a request $\mathcal{R}_a$ for HW-task τ_a^H. Contextually, τ_1 self-suspends its execution. According to Rule R3, $\mathcal{R}_a$ is inserted in the partition queue Q_1. Since partition P_1 is empty, at time $t = 1$ there is a free slot ($\eta_1(1) = 1$); hence, according to Rule R4, $\mathcal{R}_a$ is moved to Q^{FRI} and the slot of P_1 becomes reserved. Moreover, according to Rule R-P1, the FRI starts programming τ_a^H. At time $t = 5$, τ_a^H has been programmed and according to Rule R5 it starts executing.

At time $t = 1$, τ_2 starts executing being the highest-priority SW-task ready. At time $t = 2$, τ_2 concludes its sub-task $\tau_{2,1}$ by issuing a request $\mathcal{R}_c$ for τ_c^H. According to Rule R3, $\mathcal{R}_c$ is inserted in the partition queue Q_2. Since partition P_2 is empty, at time $t = 2$ there is free slot ($\eta_2(2) = 1$); hence, according to Rule R4, $\mathcal{R}_c$ is moved to Q^{FRI} and the slot of P_2 becomes reserved. However, since $\mathcal{R}_c$ has a later ticket time than $\mathcal{R}_a$, $\mathcal{R}_c$ is delayed until τ_a^H has been programmed (time $t = 5$). Then, τ_c^H can be programmed and be executed.

At time $t = 2$, τ_3 is the highest-priority SW-task ready, thus it starts executing until time $t = 3$, when it terminates its first sub-task $\tau_{3,1}$ by issuing a request $\mathcal{R}_d$ for τ_d^H. According to Rule R3, $\mathcal{R}_d$ is inserted in the partition queue Q_2. However, being the slot of P_2 busy (specifically, reserved in [5,7] and active in (7,11]), $\mathcal{R}_d$ waits in Q_2 until time $t = 11$. At time $t = 11$, τ_c^H completes its execution, Rule R6 is applied and the slot of P_2 becomes free. According to Rule R4, $\mathcal{R}_d$ is moved to Q^{FRI}, the slot of P_2 becomes again busy (specifically, reserved) and τ_d^H starts to be programmed.

Now, consider again τ_1. At time 9, τ_a^H is completed and hence the sub-task $\tau_{1,2}$ can be released. At time 10, $\tau_{1,2}$ completes by issuing a request $\mathcal{R}_b$ for HW-task τ_b^H. By Rule R3 and Rule R4, $\mathcal{R}_b$ is inserted into Q^{FRI}. Being Q^{FRI} empty, τ_b^H starts to be programmed. However, as explained above, at time $t = 11$, $\mathcal{R}_d$ (issued by τ_3) is inserted into Q^{FRI}. Being $t(\mathcal{R}_d) = 3 < t(\mathcal{R}_b) = 10$, according to Rule-R-P1 the programming of τ_b^H is preempted to program τ_d^H until time $t = 13$. Hence in [11, 13) $\mathcal{R}_b$ is delayed. Finally, note that the FRI queue is not managed in a pure FIFO manner.

D. Communication between SW and HW tasks

As stated in Section III-B, SW-tasks make use of HW-tasks to accelerate specific computations; that is, a SW-task offloads a computation to the FPGA by requesting the execution of a HW-task and then retrieves the output of such a computation to continue the execution on the processor. As shown in Figure 2, the communication between a SW-task τ_i and a HW-task τ_a^H includes two phases:

(i) sub-task $\tau_{i,j}$ prepares the input data for τ_a^H;

(ii) sub-task $\tau_{i,j+1}$ retrieves the data produced by τ_a^H.

It is worth observing that the approach used to enable such a communication can affect the real-time performance of the sys-

tem by introducing different worst-case scenarios. For instance, suppose that the output data produced by a HW-task are stored in its internal memory area and that phase (ii) comprises a copy from the local memory of the HW-task to a memory area accessible by the SW-task. In such a case, the HW-task must remain programmed onto the FPGA module until the sub-task in charge of executing the phase (ii) will be executed, otherwise output data would be lost.

Due to the scheduling delays suffered by SW-tasks, the actual time a HW-task occupies a slot is hence dependent on SW-tasks' execution behavior. Longer slot occupation times increase the delays suffered by HW-tasks, which in turn increase the delays suffered by SW-tasks by inflating their suspension time when waiting for the completion of a HW-task. Such a circular dependency can originate pathological scenarios that significantly increase the worst-case response time of SW-tasks, thus making this approach not attractive for a real-time system.

To overcome this problem, FRED adopts a different approach inspired by the capabilities of state-of-the-art platforms, where the communication between SW-tasks and HW-tasks is supported by allowing HW-tasks to directly access the shared memory $\mathcal{M}$. Hence, the two communication phases are implemented as follows:

- **(i)** sub-task $\tau_{i,j}$ prepares the input data for τ_a^H in a memory area $\mathcal{M}_a^{\text{IN}}$ inside $\mathcal{M}$, and τ_a^H retrieves the input data by directly accessing $\mathcal{M}_a^{\text{IN}}$;
- **(ii)** τ_a^H stores the output data into a memory area $\mathcal{M}_a^{\text{OUT}}$ inside $\mathcal{M}$, and $\tau_{i,j+1}$ retrieves them directly from $\mathcal{M}_a^{\text{OUT}}$, hence τ_a^H can release its slot as it finishes.

References (i.e., memory pointers) to both $\mathcal{M}_a^{\text{IN}}$ and $\mathcal{M}_a^{\text{OUT}}$ are assumed to be provided to the HW-task or known a priori. By adopting this solution, the time τ_a^H must hold a slot is totally decoupled from the scheduling delays of SW-tasks and is always upper-bounded by the WCET C_a^H plus the slot reconfiguration time r_a.

As done in most real-time analysis, bus contention times due to the interaction between HW-tasks and SW-tasks can be accounted in the WCETs. Finally, please note that such a communication approach is not limited to platforms having a main memory shared between the processor and the FPGA module, but it can also be used in platforms where a dedicated memory is reserved for such a communication. Indeed, the latter solution is more suitable for safety-critical systems requiring a higher level of predictability.

IV. Real-Time Analysis

The goal of this section is to derive a sufficient response-time analysis for the SW-tasks running under FRED. That is, for each SW-task τ_i, this section provides an upper-bound R_i on its maximum response-time such that the system is guaranteed to be schedulable if

$$\forall \tau_i \in \Gamma^S,\ R_i \leq D_i. \tag{1}$$

The upper-bounds are derived by building on Nelissen et al.'s [28] response-time analysis for *real-time fixed-segment self-suspending tasks* (SS-tasks). The SS-task model is a generic model for real-time computational activities where multiple execution phases are alternated to self-suspension phases, exactly like the execution behavior of the SW-tasks under FRED. Similarly to a SW-task, a SS-task τ_ℓ alternates the execution of $m_\ell + 1$ sub-tasks, each having WCET $C_{\ell,j}$ (j goes from 1 to $m_\ell + 1$) and m_ℓ suspension phases, each lasting *at most* $S_{\ell,j}$ time units (j goes from 1 to m_ℓ). The C, T and D parameters of a SS-task are consistently defined as the corresponding ones of a SW-task, as stated in Section III-B.

Each SW-task τ_i can hence be mapped (i.e., translated) into a SS-task τ_ℓ according to the following rules:

1) $C_{\ell,j} = C_{i,j}$, $\forall j = 1, \ldots, m_i + 1$;
2) $S_{\ell,j} = r_a + C_a^H + \Delta_a$, $\forall j = 1, \ldots, m_i$, where $\tau_a^H \in \mathcal{H}(\tau_i)\ :\ \tau_i := \langle \ldots, \tau_{i,j}, \tau_a^H, \tau_{i,j+1}, \ldots \rangle$.
3) unless differently specified, $\mathcal{X}_\ell = \mathcal{X}_i$, where $\mathcal{X}_i$ is a parameter of τ_i.

Intuitively speaking, Rule 1 maps each sub-task of the SW-task τ_i into a sub-task of the SS-task τ_ℓ, while Rule 2 defines a suspension phase of the SS-task for each HW-task τ_a^H used by τ_i; such a suspension phase includes the reconfiguration time r_a, the WCET C_a^H and the the worst-case delay Δ_a suffered by the HW-task under the FRED scheduling infrastructure. Finally, Rule 3 enforces that the other parameters of the SS-task τ_ℓ are equal to the ones of the SW-task τ_i.

Please note that all the parameters mentioned in the rules above are known, except for the delay Δ_a: computing a safe upper-bound on such a delay is the main challenge of the proposed real-time analysis.

For the sake of completeness, the next section briefly summarizes the Nelissen et al.'s response-time analysis for SS-tasks.

A. Summary on Nelissen et al.'s analysis

The response-time analysis of fixed-priority SS-tasks is a problem studied since several years in the real-time community. However, Nelissen et al. [28] discovered several errors in many papers concerning the analysis of SS-tasks, proving that most of the published results are not safe and proposing a safe and accurate response-time analysis for SS-tasks based on *mixed-integer linear programming* (MILP).

A MILP formulation is instantiated for each SS-task τ_ℓ whose objective is to *maximize* the response-time upper-bound $R_{\ell,j}$ of each sub-task $\tau_{\ell,j}$ composing τ_ℓ. Each response-time upper-bound is expressed as $R_{\ell,j} = C_{\ell,j} + \sum_{\tau_k \in hp(\tau_\ell)} \sum_{z=1}^{m_k} NI_{\ell,k,z} \times C_{k,z}$ where $hp(\tau_\ell)$ is the set of tasks that have higher priority than τ_ℓ and $NI_{\ell,k,z}$ is an *integer optimization variable* modeling the number of jobs of $\tau_{\ell,j}$ interfering with τ_ℓ. Several constraints are enforced to bound the value of $NI_{\ell,k,z}$: please refer to [28] for further details. Finally, once the MILP has been solved, the total response-time upper-bound R_ℓ of τ_ℓ is computed as $R_\ell = \sum_{j=1}^{m_\ell} R_{\ell,j} + \sum_{j=1}^{m_\ell - 1} S_{\ell,j}$.

B. Upper-bound for the delay Δ_a

As stated above, computing a response-time upper-bound for SW-tasks is crucial to bound the maximum time a SW-task can be suspended to wait for the completion of a HW-task. This in turn requires bounding the delay Δ_a suffered by each HW-task request $\mathcal{R}_a$, which is the goal of this section.

As a first step, the following lemma establishes that the ticket-based scheduling policy introduced in Section III-C is *work-conserving*.

Lemma 1: A HW-task request $\mathcal{R}_a$ for τ_a^H with affinity to partition $P_k = P(\tau_a^H)$ is delayed at time t if and only if either

- *all* the n_k^S slots of P_k are *busy* serving other HW-tasks $\tau_b^H \neq \tau_a^H$ with $P(\tau_b^H) = P_k$; or
- the FRI is busy programming other HW-tasks $\tau_b^H \neq \tau_a^H$.

Proof: A HW-task request $\mathcal{R}_a$ can be delayed either (i) when it is in the partition queue Q_k or (ii) when it is in the Q^{FRI} queue. In case (i), according to Rule 4, $\mathcal{R}_a$ can wait as long as *all* the n_k^S slots of P_k are *busy*. In case (ii), we distinguish between preemptive and non-preemptive FRI management. Under preemptive FRI management, being Q^{FRI} non-empty (at least $\mathcal{R}_a$ is inside Q^{FRI}), by Rule R-P1 the FRI is still programming a HW-task τ_a^H in a reserved slot in $P(\tau_a^H)$. By Rule R4, for every request inserted into Q^{FRI} there is a reserved slot in the corresponding partition. The same argument holds under non-preemptive FRI management by considering Rules R-NP1 and R-NP2. ∎

By relying on the fact that SW-tasks issue HW-task requests in a sequential fashion, it is possible to establish another key property.

Lemma 2: Let $\mathcal{R}$ be an arbitrary HW-task request issued by a SW-task τ_i at time $t(\mathcal{R})$ and let $t_s(\mathcal{R})$ be the time at which $\mathcal{R}$ is removed from Q^{FRI} (i.e., $\mathcal{R}$ is satisfied). Let also $pend(\mathcal{R})$ be the set of pending HW-task requests during $[t(\mathcal{R}), t_s(\mathcal{R}))$ that have earlier (or equal) ticket time, i.e., $\forall \mathcal{R}_a \in pend(\mathcal{R}),\ t(\mathcal{R}_a) \leq t(\mathcal{R})$. If HW-task requests are serialized as specified in the model presented in Section III-B, then each SW-task $\tau_j \neq \tau_i$ can possibly issue *at most one* HW-task request in $pend(\mathcal{R})$.

Proof: By contradiction. Suppose that $pend(\mathcal{R})$ contains more than one HW-task request from SW-task τ_j, say $\mathcal{R}_a$ and $\mathcal{R}_b$, respectively issued at times $t(\mathcal{R}_a)$ and $t(\mathcal{R}_b)$ and satisfied at times $t_s(\mathcal{R}_a)$ and $t_s(\mathcal{R}_b)$. Without loss of generality assume $t(\mathcal{R}_a) \leq t(\mathcal{R}_b)$. By definition of set $pend(\mathcal{R})$ we have $t(\mathcal{R}_a) \leq t(\mathcal{R}_b) \leq t(\mathcal{R})$, $t_s(\mathcal{R}_a) > t(\mathcal{R})$ and $t_s(\mathcal{R}_b) > t(\mathcal{R})$. Hence we obtain $t(\mathcal{R}_a) \leq t(\mathcal{R}_b) \leq t(\mathcal{R}) < t_s(\mathcal{R}_a)$, which implies that $\mathcal{R}_b$ has been issued before the completion of $\mathcal{R}_a$. This contradicts the assumption, according to which HW-task requests issued by the same SW-task are serialized. Hence, the lemma follows. ∎

We are now ready to derive the upper-bound for the delay Δ_a. To this end, it is necessary to distinguish between preemptive and non-preemptive management of the FRI.

1) Preemptive FRI management:

Theorem 1: Consider an arbitrary HW-task request $\mathcal{R}_a$ for τ_a^H issued by a SW-task τ_i. Let $P_k = P(\tau_a^H)$ be the affinity of τ_a^H. Under *preemptive* management of the FRI, the maximum delay Δ_a incurred by $\mathcal{R}_a$ is upper-bounded by

$$\overline{\Delta_a^P} = \sum_{\tau_j \neq \tau_i} \max_{\tau_b^H \in \mathcal{H}(\tau_j)} \left\{ \Delta_b^{slot} + r_b \right\} \tag{2}$$

where

$$\Delta_b^{slot} = \begin{cases} \frac{C_b^H}{n_k^S} & \text{if } P(\tau_b^H) = P_k \\ 0 & \text{otherwise.} \end{cases}$$

Proof: Let $\mathcal{X}$ be the set of HW-task requests that delay $\mathcal{R}_a$. Since we are considering preemptive management of the FRI, Rule R-P1 applies. Because of such a rule and the FIFO ordering of the partition queue Q_k, $\mathcal{R}_a$ can only be delayed by other requests that have *earlier* (or equal) ticket time. Hence, $\forall \mathcal{R} \in \mathcal{X},\ t(\mathcal{R}) \leq t(\mathcal{R}_a)$.

To help the presentation we define the set of HW-tasks $\Gamma_{\mathcal{X}}^H$ by mapping each HW-task request $\mathcal{R} \in \mathcal{X}$ into the corresponding HW-task $\tau^H \in \Gamma_{\mathcal{X}}^H$. By Lemma 2, each SW-task $\tau_j \neq \tau_i$ can have issued at most one HW-task request in $\mathcal{X}$. Hence, $\forall \tau_i \neq \tau_j, |\Gamma_{\mathcal{X}}^H \cap \mathcal{H}(\tau_j)| \leq 1$. Moreover, since one HW-task cannot be used by multiple SW-tasks (i.e., $\bigcap_{\tau_j \in \Gamma^S} \mathcal{H}(\tau_j) = \emptyset$), each request in $\mathcal{X}$ corresponds to *one and only one* HW-task in $\Gamma_{\mathcal{X}}^H$, hence $|\Gamma_{\mathcal{X}}^H| = |\mathcal{X}|$.

The total workload W that can delay $\mathcal{R}_a$ cannot be greater than the sum of **(i)** the execution time of HW-tasks in $\Gamma_{\mathcal{X}}^H$ and **(ii)** the reconfiguration time of HW-tasks in $\Gamma_{\mathcal{X}}^H$. In addition, by Lemma 1, $\mathcal{R}_a$ cannot be delayed by the execution of HW-tasks τ_b^H with $P(\tau_b^H) \neq P_k$. Hence, we have

$$W \leq \sum_{\substack{\tau_b^H \in \Gamma_{\mathcal{X}}^H \\ P(\tau_b^H) = P_k}} \left(C_b^H + r_b \right) + \sum_{\substack{\tau_b^H \in \Gamma_{\mathcal{X}}^H \\ P(\tau_b^H) \neq P_k}} r_b,$$

and being HW-task requests scheduled in a work-conserving manner (by Lemma 1), $\Delta_a \leq W$ holds.

Now, note that any interval in which $\mathcal{R}_a$ is delayed can be considered as an alternating sequence of sub-intervals of two types:

- **(i)** *type X*: if Q^{FRI} contains at least a request with a ticket time less than $t(\mathcal{R}_a)$, i.e., $\exists \mathcal{R} \in Q^{FRI} \ :\ t(\mathcal{R}) \leq t(\mathcal{R}_a)$);
- **(ii)** *type Y*: otherwise, i.e., $\nexists \mathcal{R} \in Q^{FRI} \ :\ t(\mathcal{R}) \leq t(\mathcal{R}_a)$.

Being such intervals complementary, this classification is well defined.

Let Δ^X be the total delay suffered by $\mathcal{R}_a$ during intervals of type X and let Δ^Y be the one suffered during intervals of type Y. Hence, the total delay is given by $\Delta_a = \Delta^X + \Delta^Y$. To derive a bound on Δ_a, we proceed by deriving a bound on each of these two terms.

Type X) Consider an arbitrary time instant during an interval of type X. Let $\mathcal{R}$ be the request at the head of Q^{FRI}. By definition of type X interval and Rule R2, $t(\mathcal{R}) \leq t(\mathcal{R}_a)$: hence $\mathcal{R}$ contributes to the workload W. According to Rule R-P1, the FRI is programming the HW-task related to $\mathcal{R}$, and hence $\mathcal{R}_a$ is delayed anyhow by such an operation.

In total, $\mathcal{R}_a$ cannot be delayed by more than the overall reconfiguration times in the workload W, hence $\Delta^X \leq \sum_{\tau_b^H \in \Gamma_{\mathcal{X}}^H} r_b$.

Type Y) In this case, the queue Q^{FRI} can be *empty* or *non-empty*. If Q^{FRI} is empty, being $\mathcal{R}_a$ delayed, it must be waiting into its partition queue Q_k. If Q^{FRI} is non-empty, $\mathcal{R}_a$ cannot be into Q^{FRI} otherwise it would not be delayed according to Rule R2 and Rule R-P1 (i.e., it would be at the head of Q^{FRI}); hence $\mathcal{R}_a$ is waiting in Q_k anyway.

According to Rule 4, if $\mathcal{R}_a$ waits into Q_k, then all the n_k^S slots of P_k are busy. Moreover, none of these slots can be reserved: this holds because of the FIFO ordering of Q_k (in fact no request with affinity P_k can be into Q^{FRI}). Hence, in this case, *all* the n_k^S slots of P_k are *active* serving the execution of HW-tasks τ_b^H with $P(\tau_b^H) = P_k$. As a consequence, if $\mathcal{R}_a$ is delayed by Δ^Y time units across all intervals of type Y, then partition P_k served $n_k^S \cdot \Delta^Y$ execution time units.

By looking at the workload W that can interfere with $\mathcal{R}_a$, it must be that

$$n_k^S \cdot \Delta^Y \leq \sum_{\substack{\tau_b^H \in \Gamma_{\mathcal{X}}^H \\ P(\tau_b^H)=P_k}} C_b^H$$

and hence

$$\Delta^Y \leq \sum_{\substack{\tau_b^H \in \Gamma_{\mathcal{X}}^H \\ P(\tau_b^H)=P_k}} \frac{C_b^H}{n_k^S}.$$

Rewriting the expression for Δ_a we obtain

$$\Delta_a = \Delta^X + \Delta^Y \leq \sum_{\substack{\tau_b^H \in \Gamma_{\mathcal{X}}^H \\ P(\tau_b^H)=P_k}} \left(\frac{C_b^H}{n_k^S} + r_b \right) + \sum_{\substack{\tau_b^H \in \Gamma_{\mathcal{X}}^H \\ P(\tau_b^H)\neq P_k}} r_b. \quad (3)$$

The upper-bound on Δ_a follows by maximizing Equation (3) over all possible sets $\Gamma_{\mathcal{X}}^H$. Since by construction of $\Gamma_{\mathcal{X}}^H$ each SW-task $\tau_j \neq \tau_i$ can have at most one request for a HW-task $\tau_b^H \in \Gamma_{\mathcal{X}}^H$, Equation (2) accounts for the maximum contribution to Equation (3) given by each SW-task $\tau_j \neq \tau_i$. ■

2) Non-preemptive FRI management: Building on the bound $\overline{\Delta_a^P}$ stated by Theorem 1, it is possible to derive a bound on the delay incurred in the case of non-preemptive FRI.

Theorem 2: Consider an arbitrary HW-task request $\mathcal{R}_a$ for τ_a^H issued by a SW-task τ_i. Let $P_k = P(\tau_a^H)$ be the affinity of τ_a^H. Under *non-preemptive* management of the FRI, the maximum delay Δ_a incurred by $\mathcal{R}_a$ is upper-bounded by

$$\overline{\Delta_a^{NP}} = \overline{\Delta_a^P} + NH_k^{max} \times r_k^{max} \quad (4)$$

where

$$NH_k^{max} = \left| \{ \tau_b^H \in \Gamma^H \ : \ P(\tau_b^H) = P_k \} \right|$$

and

$$r_k^{max} = \max_{\tau_b^H \in \Gamma^H} \{ r_b \ : \ P(\tau_b^H) \neq P_k \}.$$

Proof: Any interval in which $\mathcal{R}_a$ is delayed can be considered as an alternating sequence of sub-intervals of two types:

- **(i)** *type X*: if Q^{FRI} contains at least a request corresponding to a HW-task τ_b^H with $P(\tau_b^H) = P_k$ (i.e., same partition of τ_a^H) waiting into Q^{FRI};
- **(ii)** *type Y*: otherwise.

Note that these intervals have a different definition with respect the ones used in the proof of Theorem 1, but the same properties apply (i.e., the classification is well defined). We proceed by considering each type separately.

Type X) Consider a single interval of type X. Because of the FIFO ordering of partition queues Q_k, whenever $\mathcal{R}_a$ is delayed (during the considered interval), it must be waiting for the reconfiguration of HW-tasks with affinity P_k (present into Q^{FRI} by definition). According to Rule R-NP1 and Rule R-NP2, their corresponding request (or $\mathcal{R}_a$ itself) can be delayed by **(i)** other requests with lower (or equal) ticket time that are into Q^{FRI}; and **(ii)** *at most one* request with *higher* ticket time which is (non-preemptively) served by the FRI. Because of the FIFO ordering of partition queue Q_k, this latter request must be related to a HW-task with affinity $\neq P_k$. In case (i), the same consideration argued in the proof of Theorem 1 holds (preemptive FRI). In case (ii), the delay suffered by $\mathcal{R}_a$ cannot be higher than $r_k^{max} = \max_{\tau_b^H \in \Gamma^H} \{ r_b \ : \ P(\tau_b^H) \neq P_k \}$ time units. By Rule R-NP2, such a delay can occur at most once for each interval of type X.

The total number of intervals of type X is maximized when, during each of such intervals, there is only *one* request into Q^{FRI} that corresponds to a HW-task with affinity P_k. Hence, such a number is bounded by the total number of HW-tasks with affinity P_k, given by $NH_k^{max} = |\{ \tau_b^H \in \Gamma^H \ : \ P(\tau_b^H) = P_k \}|$. Hence, the total delay in case (ii) across all intervals of type X is upper-bounded by $NH_k^{max} \times r_k^{max}$.

Type Y) By definition, Q^{FRI} contains no requests corresponding to HW-tasks with the same affinity of τ_a^H. This clearly implies that $\mathcal{R}_a$ is either completed (and hence not yet delayed) or waiting inside Q_k. This is the same situation discussed in the proof of Theorem 1 when considering the term Δ^Y.

In summary, the total delay suffered by $\mathcal{R}_a$ in intervals of type Y, plus the delay in case (i) during intervals of type X, is bounded by the upper-bound $\overline{\Delta_a^P}$ of the delay suffered under preemptive FRI management. Hence, the theorem follows. ■

V. Practical validation and profiling

This section presents a preliminary case study implemented on the Zynq-7000 platform to evaluate the feasibility of the proposed approach, profile hardware acceleration speedup factors, and measure reconfiguration overheads. The considered platform includes a dual-core ARM Cortex-A9 processor and a 7-series FPGA integrated on the same chip. The internal structure of a Zynq SoC can be divided in two main functional blocks referred to as processing system (PS) and programmable logic (PL) [14]. The PS block comprises the ARM Cortex-A9 MPCore, the memory interfaces and the I/O peripherals, while the PL block includes the FPGA programmable fabric. The subsystems included in the PS are interconnected among themselves and to the PL through an ARM AMBA AXI (Advanced eXtensible Interface) interconnect.

The hardware modules configured on the PL can access the interconnect through a set of master and slave AXI interfaces exported by the PS side to the PL side. Slave interfaces allow modules to access the global memory space and share the DRAM memory with the processors. Dynamic partial reconfiguration is supported under the PS control. PL fabric can be fully or partially (re)configured by the PS through the device configuration interface (DevC) subsystem. The DevC includes a DMA engine that can be programmed to transfer bitstreams from the main memory to the PL configuration memory through the processor configuration access port (PCAP).

A. System architecture

In the prototype developed for the case study, the PL area is divided in two main regions: a static region and a reconfigurable region. The static region contains the communication infrastructure and other support modules, while the reconfigurable region is organized as a single partition divided into S slots, each hosting a HW-task.

In general, since bitstream relocation is not supported by the Xilinx standard tools [25] [26], each HW-task τ_i^H is

implemented as a set of n_k^S bitstreams, one for each slot S_j of its associated partition $P(\tau_i^H)$. Each slot S_j can accommodate all the specific implementations of each HW-task τ_i^H that belongs to partition $P(\tau_i^H)$.

Since the slot interface should match the one of the HW-tasks [26], we have defined a common interface that all HW-tasks are required to implement. Such a common interface is similar to the one adopted by Sadri et al. [29]. The interface includes an AXI master interface for accessing the system memory, an AXI slave interface through which the HW-task can be controlled by the PS, and an interrupt signal to notify the PS. The AXI master interface logic allows HW-tasks to retrieve data autonomously from the memory space, implementing the communication mechanism described in Section III-D.

In the current experimental setup, the AXI master interfaces exported by the HW-tasks are attached to high-performance slave ports exported by the PS, while the AXI slave control interfaces are attached to the general purpose master ports. The software part consists of a user-level library for the FreeRTOS operating system. The library abstracts the reconfiguration mechanism and provides a simple API that enables SW-tasks to request the execution of HW-tasks on the PL through the `EXECUTE_HW_TASK()` function, as described in Figure 2.

B. Experimental setup

The prototype has been deployed on a ZYBO board, featuring the Z-7010 Zynq SoC supported by 512 MB of DDR3 memory. The ARM cores included in the PS run at 650 MHz while the clock frequency for the PL is set to 100 MHz. In this experimental setup the single reconfigurable partition has been divided in two slots, each containing about 25% of the *slices* available in the programmable logic. The remaining 50% of the resources are allocated to the static part. Since both slots have the same dimensions, also the partial bitstreams resulting from the logic synthesis process have the same size of 338 KByte. Therefore, a large number of partial bitstreams can be stored in the 512 MB RAM memory.

The developed case study includes four standard functions implemented both as HW-tasks and software code: three simple image convolution filters (Sobel, Blur, and Sharp) and a matrix multiplier. The HW-tasks have been designed with the Xilinx Vivado ® high-level synthesis tool, while the software versions have been implemented in C language. The image processing HW-tasks process images of size 800×600 pixels, with 24-bit color depth. The matrix multiplier HW-task has been synthesized to multiply 512 elements integer matrices.

C. Experimental results

1) Speedup evaluation experiment: A first experiment has been carried out to measure the speedup factors achievable from hardware acceleration on FPGA. The *longest observed execution times* (LOET) of the four HW-tasks have been measured and compared against the execution times of their software counterparts over 1000 runs. The results are reported in Table III. The minimum speedup has been computed as the ratio between the minimum software execution time and the maximum hardware execution time observed. Despite the clock frequency (100 MHz) of the FPGA was slower than the one of the processor (650 MHz), hardware accelerated implementations provided a relevant speedup between 5 and 15 over their software counterparts.

Operation	FPGA LOET [ms]	Software LOET [ms]	Min speedup
Sobel	19.763	178.874	9.050
Blur	24.629	374.164	15.190
Sharp	24.630	306.539	12.386
Mult	1696.327	8774.103	5.170

Table III
SPEEDUP EVALUATION.

2) Response time experiment: A second experiment was carried out to evaluate the longest observed response times in a scenario where the number of HW-tasks exceeds the number of slots. The task set used for this test includes four SW-tasks that use the four HW-tasks defined in Section V-B. SW-tasks are assigned priorities according to the rate-monotonic algorithm. Table IV summarizes the task parameters and the longest observed response times in a 8-hour run.

Considering the software execution times profiled in the previous experiment, it is worth noticing that the task set used for this experiment can only be scheduled through hardware acceleration. The equivalent task set, implemented as purely software tasks, with the same periods and priorities, it is clearly not schedulable on the processor.

Task	Period [ms]	Longest Observed Response Time [ms]
Sobel	100	43.748
Blur	150	69.438
Sharp	170	74.855
Mult	2500	1723.200

Table IV
LONGEST OBSERVED RESPONSE TIMES.

Figure 6 shows the distribution of the reconfiguration times observed in the previous scheduling test for more than 500,000 reconfiguration events. The longest observed reconfiguration time is 2.845 ms. Therefore, given the size of the bistreams, the minimum observed throughput of the PCAP configuration port resulted to be 116 MB/s, which is consistent with the maximum throughput of 145 MB/s stated by Xilinx [14].

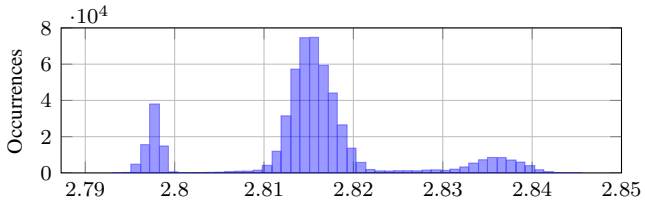

Figure 6. Distribution of reconfiguration times (ms).

It is worth mentioning that the set of four HW-tasks used for this experiment cannot be implemented statically using only 50% of the resources available on the the PL fabric. Partial reconfiguration allows to virtually extend the number of resources to accommodate all the HW-tasks in timesharing.

Overall, this case study has shown that, despite reconfiguration times are not negligible, the proposed approach can be implemented using current FPGA technology to improve the performance of real-time applications with respect to a pure software implementation.

VI. Experimental results

This section presents a set of experiments aimed at evaluating the performance of the FRED scheduling infrastructure in terms of schedulability analysis with synthetic workload. The experiments are performed to verify the schedulability under different architecture configurations and are obtained applying the sufficient response time analysis presented in Section IV.

A. Task set and architecture generation

The synthetic workload has been generated as follows.

1) Hardware architecture: The FRI throughput is defined as $\rho = 100$ and the total number of logic blocks is set as $b = 1,000,000$: the ratio between these two values yields an FRI throughput similar to the one observed in Section V. The logic blocks of the FPGA are equally distributed among the partitions. The same holds for the logic blocks of each partition, which are equally distributed among its slots.

2) SW and HW-tasks: For simplicity, we focus on the case where each SW-task accesses a single HW-task, i.e., $m_i = 2, \forall \tau_i \in \Gamma^S$. Because of this restriction, we denote HW-tasks with the same index of the corresponding SW-task (i.e., τ_i^H is the HW-task used by SW-task τ_i). For each partition P_k, a bucket of possible task periods is defined according to the following rules: (i) the interval of periods covered by any two buckets do not overlap; (ii) all the values in every bucket are in the range $\left[10^5, 10^6\right] \mu s$. For each SW-task τ_i, a random period T_i is chosen from the bucket corresponding to partition $P(\tau_i^H)$ and then removed from the bucket. The UUniFast algorithm [30] is used to generate the utilization factor U_i of each SW-task τ_i, such that $\sum_{\tau_i \in \Gamma^S} U_i = U$, where U is parameter defined in the experiments. The minimum task utilization is set to $U^{min} = 0.005$. The WCET of each SW-task τ_i is then computed as $C_i = U_i \cdot T_i$. Such a value has been then randomly split to obtain the WCETs $C_{i,1}$ and $C_{i,2}$ of each sub-task. Finally, a parameter U^H has been defined to "mimic" a notion of utilization of the FPGA (also referred to as *hardware utilization*). Note that, due to the intrinsic interaction between SW- and HW-tasks, this parameter cannot be related to a pure concept of utilization like the parameter U. Again, the UUniFast algorithm [30] algorithm was used to generate the hardware utilization for each hardware task as follows: $\forall \tau_i^H \in \Gamma^H, C_i^H = U_i^H \cdot T_i$. Like U, U^H is another parameter varied in the experiments. Task priorities π_i are assigned according to the rate-monotonic policy.

B. Experiments on schedulability analysis

This set of experiments has been carried out to measure the schedulability ratio of the tested task sets under four different configurations:

- **(i)** *Static*: the FPGA is supposed to have an infinite area, so that all the HW-tasks are statically assigned to a fixed slot, thus reconfiguration is not needed.
- **(ii)** *FRED-P*: the proposed approach is used with preemptive FRI management and scheduling delays are computed according to Theorem 1;
- **(iii)** *FRED-NP*: same as FRED-P but with non-preemptive FRI management and delays computed by Theorem 2;
- **(iv)** *Software*: all task sets are implemented in software. Worst-case execution times of the resulting software implementation of HW-tasks are computed considering a speed-up factor Φ, assumed to be the same for all the HW-tasks.

A first experiment varied the utilization U of a fixed number of tasks (both HW and SW), using $n_P = 3$ partitions, each with $n_k^S = 2$ slots, and 3 HW-tasks with affinity to each partition. The total number of tasks n_S has been chosen to overload the system: each partition P_k has $n_k^S + 1$ tasks, hence *all the tested task sets are not feasible without enabling DPR*. Figure 7 reports the results of an experiment where U was varied from 0.05 to 0.95 with a step of 0.05, with $\Phi = 1$ and $U^H = 0.1$. Please keep in mind that the small value of U^H cannot be interpreted as in the classical software semantics.

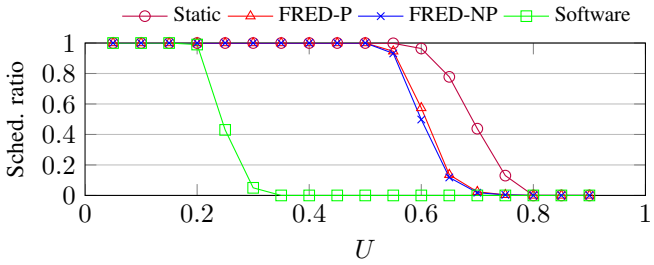

Figure 7. Schedulability ratio as a function of U.

As evident from the plots, the *Software* approach quickly degrades at utilization values that are much lower than the ones at which *FRED-P* and *FRED-NP* starts being no more able to schedule the task sets. Even with an *unrealistic* and limit-case value for the speed-up ($\Phi = 1$), this happens because the FPGA allows for intrinsic parallelism with respect to a single processor. The small difference between *FRED-P* and *FRED-NP* is due to the fact that the reconfiguration time (chosen according the profiling of Section V) results almost negligible with respect to the generated execution times of HW-tasks. The *Static* approach represents a theoretical upper-bound (i.e, not practically achievable because assumes FPGAs with infinite area), whose performance depends on the absence of reconfiguration times and contention for slots and the FRI. In this experiment FRED obtains a schedulability ratio higher than 50% up to $U = 0.6$, outperforming the pure *Software* approach. Moreover, the results are quite close to the ideal scenario provided by a fully *Static* approach.

Figure 8 reports the result of another experiment where the the software utilization has been fixed to $U = 0.1$ and U^H has been varied from 0.05 to 0.95 with a step of 0.05. Also in this case, FRED outperforms pure *Software* approach, guaranteeing more than 50% of the tested task sets up to $U^H = 0.4$.

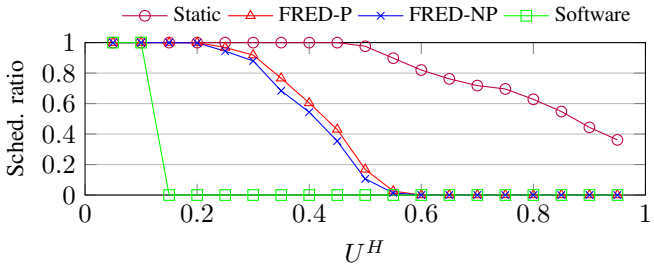

Figure 8. Schedulability ratio as a function of U^H.

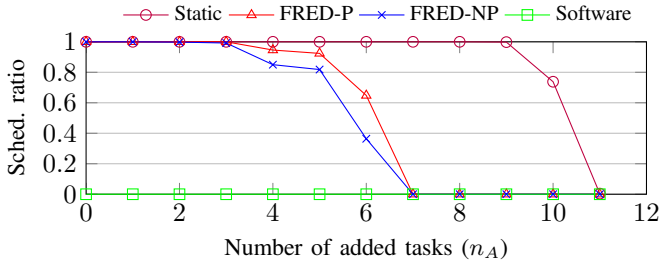

Figure 9. Schedulability ratio as a function of the number n_A of added tasks.

A third experiment has been carried out to investigate the benefits of FRED for applications that fully saturate the FPGA area and hence cannot be extended to include additional tasks without exploiting DPR. We considered systems with 2 partitions, 2 slots per partition and $\Phi = 3$. Starting from a fixed scenario ($U^H = 0.1$, $U = 0.1$) where all the slots are occupied by HW-tasks, Figure 9 reports the schedulability ratio obtained by *adding n_A new tasks* (each consisting of a SW-task and the corresponding HW-task), where n_A was varied from 0 to 12. Each added task τ_i has $U_i^H = 0.05$ and $U_i = 0.05$, while the other parameters were generated as in the previous experiments. The affinity of each new task was chosen as specified in Section VI-A2. This experiment clearly shows that FRED allows guaranteeing real-time applications extended with *more than 5 tasks*, which otherwise could not be executed.

VII. Conclusions

This paper presented a framework for supporting the development of safety-critical real-time applications on computing platforms that include a processor and an FPGA module with dynamic partial reconfiguration capabilities.

After providing a model of the platform and the computational activities, a scheduling infrastructure was proposed to bound the delays experienced by the tasks, and a response-time analysis was derived to verify the schedulability of safety-critical applications with real-time constraints. The approach has also been implemented and validated on the Zynq platform to demonstrate its practical applicability. The experimental results performed on synthetic workloads showed the performance of the analysis in different scenarios.

As a future work, we plan to incorporate the proposed framework inside the Erika Enterprise operating system, providing specific system call that simplify the development of safety-critical real-time applications on heterogeneous computing platforms using FPGA accelerated components.

Acknowledgements

The authors like to thank Geoffrey Nelissen for his valuable support in setting up the analysis for self-suspending tasks.

References

[1] S. Habinc, "Technical report: Suitability of reprogrammable FPGAs in space applications," Gaisler Research, Tech. Rep., September 2002.

[2] P. Alfke, "Recent progress in field programmable logic," in *Proceedings of the 6th Workshop on Electronics for LHC Experiments*, 2000.

[3] M. Goosman, N. Dorairaj, and E. Shiflet. (2006) How to take advantage of partial reconfiguration in FPGA designs. [Online]. Available: www.eetimes.com/document.asp?doc_id=1274489

[4] S. Liu, R. N. Pittman, and A. Forin, "Minimizing partial reconfiguration overhead with fully streaming dma engines and intelligent ICAP controller," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, February 2010.

[5] *Partial Reconfiguration User Guide*, Xilinx, 2012, v14.1.

[6] F. Duhem, F. Muller, and P. Lorenzini, "Farm: Fast reconfiguration manager for reducing reconfiguration time overhead on FPGA," in *Proceedings of the 7th International Conference on Reconfigurable Computing: Architectures, Tools and Applications*, November 2011.

[7] M. Pagani, M. Marinoni, A. Biondi, A. Balsini, and G. Buttazzo, "Towards real-time operating systems for heterogeneous reconfigurable platforms," in *Proc. of the 12th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2016)*, 2016.

[8] S. D. Scott, A. Samal, and S. Seth, "Hga: A hardware-based genetic algorithm," in *Proceedings of the ACM Third International Symposium on Field-programmable Gate Arrays*, February 1995.

[9] W. Chen, P. Kosmas, M. Leeser, and C. Rappaport, "An FPGA implementation of the two-dimensional finite-difference time-domain (fdtd) algorithm," in *Proceedings of the ACM/SIGDA 12th international symposium on Field programmable gate arrays*, February 2004.

[10] Digilent. (2016) Zybo reference manual. [Online]. Available: https://reference.digilentinc.com/_media/zybo/zybo_rm.pdf

[11] E. Martins, L. Almeida, and J. A. Fonseca, "An FPGA-based coprocessor for real-time fieldbus traffic scheduling: Architecture and implementation," *Journal of Systems Architecture*, vol. 51, no. 1, pp. 29–44, 2005.

[12] M. D. Natale and E. Bini, "Optimizing the FPGA implementation of hrt systems," in *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS)*, April 2007.

[13] I. Gray, Y. Chan, J. Garside, N. Audsley, and A. Wellings, "Transparent hardware synthesis of java for predictable large-scale distributed systems," in *Proceedings of the Second International Workshop on FPGAs for Software Programmers (FSP)*, September 2015.

[14] *Zynq-7000 AP SoC Technical Reference Manual*, Xilinx, 2015, v1.10.

[15] R. Pellizzoni and M. Caccamo, "Real-time management of hardware and software tasks for FPGA-based embedded systems," *IEEE Transactions on Computers*, vol. 56, no. 12, pp. 1666–1680, December 2007.

[16] K. Danne and M. Platzner, "Periodic real-time scheduling for FPGA computers," in *Proceedings of the 3rd International Workshop on Intelligent Solutions in Embedded System*, May 2005.

[17] S. Saha, A. Sarkar, and A. Chakrabarti, "Scheduling dynamic hard real-time task sets on fully and partially reconfigurable platforms," *IEEE Embedded Systems Letters*, vol. 7, no. 1, pp. 23–26, March 2015.

[18] F. Dittmann and S. Frank, "Hard real-time reconfiguration port scheduling," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, April 2007.

[19] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, "Controller area network (CAN) schedulability analysis: Refuted, revisited and revised," *Real-Time System*, vol. 35, no. 3, pp. 239–272, 2007.

[20] E. Lübbers and M. Platzner, "Reconos: Multithreaded programming for reconfigurable computers," *ACM Transactions on Embedded Computing Systems*, vol. 9, no. 1, pp. 8:1–8:33, October 2009.

[21] ——, "Cooperative multithreading in dynamically reconfigurable systems." in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, August 2009.

[22] M. Happe, A. Traber, and A. Keller, *Proceedings of the 11th International Symposium on Applied Reconfigurable Computing (ARC)*. Springer International Publishing, April 2015, ch. in Preemptive Hardware Multitasking in ReconOS, pp. 79–90.

[23] X. Iturbe, K. Benkrid, C. Hong, A. Ebrahim, R. Torrego, and T. Arslan, "Microkernel architecture and hardware abstraction layer of a reliable reconfigurable real-time operating system (R3TOS)," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 8, no. 1, pp. 5:1–5:35, March 2015.

[24] C. Beckhoff, D. Koch, and J. Torresen, "Go ahead: A partial reconfiguration framework," in *Proc. of the 20th Annual IEEE Int. Symposium on Field-Programmable Custom Computing Machines*, April 2012.

[25] D. Koch, *Partial Reconfiguration on FPGAs: Architectures, Tools and Applications*. Springer-Verlag New York, February 2012.

[26] *Vivado Design Suite User Guide: Partial Reconfiguration*, Xilinx, 2015, v2015.4.

[27] J. Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip," in *Proceedings of DATE*, March 2003.

[28] G. Nelissen, J. Fonseca, G. Raravi, and V. Nelis, "Timing analysis of fixed priority self-suspending sporadic tasks," in *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, July 2015.

[29] M. Sadri, C. Weis, N. Wehn, and L. Benini, "Energy and performance exploration of accelerator coherency port using xilinx zynq," in *Proceedings of the 10th FPGAworld Conference*, September 2013.

[30] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1-2, pp. 129–154, May 2005.