



HAL
open science

A Linux-Based Support for Developing Real-Time Applications on Heterogeneous Platforms with Dynamic FPGA Reconfiguration

Marco Pagani, Alessandro Biondi, Mauro Marinoni, Lorenzo Molinari,
Giuseppe Lipari, Giorgio Buttazzo

► **To cite this version:**

Marco Pagani, Alessandro Biondi, Mauro Marinoni, Lorenzo Molinari, Giuseppe Lipari, et al.. A Linux-Based Support for Developing Real-Time Applications on Heterogeneous Platforms with Dynamic FPGA Reconfiguration. *Future Generation Computer Systems*, 2022, 129, pp.125-140. 10.1016/j.future.2021.11.007 . hal-03512476

HAL Id: hal-03512476

<https://hal.science/hal-03512476v1>

Submitted on 5 Jan 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Linux-Based Support for Developing Real-Time Applications on Heterogeneous Platforms with Dynamic FPGA Reconfiguration*

Marco Pagani^{a,*}, Alessandro Biondi^a, Mauro Marinoni^a, Lorenzo Molinari^a, Giuseppe Lipari^b, Giorgio Buttazzo^a

^a*Scuola Superiore Sant'Anna, Pisa*

^b*Université de Lille, CNRS, Centrale Lille, UMR 9189 – CRISTAL*

Abstract

Computing platforms for next-generation cyber-physical systems are evolving towards heterogeneous architectures comprising different processing elements and hardware accelerators. In particular, SoC-FPGA platforms, including multiple general-purpose processing cores tightly coupled with an FPGA fabric, represent an attractive solution due to their flexibility, efficiency, and timing predictability. On these platforms, dedicated hardware accelerators implemented on the FPGA fabric can offload computationally intensive activities from general-purpose processing cores. Furthermore, dynamic partial reconfiguration allows virtualizing the FPGA resources by sharing them among multiple hardware accelerators over time.

Although very promising, FPGA-based hardware acceleration also introduces new challenges, such as managing and scheduling multiple concurrent acceleration and reconfiguration requests. The FRED framework has been proposed to address these challenges while preserving the predictability required by real-time systems. FRED is based on a device model that matches the capabilities of contemporary SoC-FPGA platforms and comes with an ad-hoc scheduling infrastructure designed to guarantee bounded response times for DPR-enabled accelerated tasks. This paper presents Fred-Linux, the reference implementation of the FRED framework for GNU/Linux. Fred-Linux allows developing rich applications while leveraging predictable FPGA-based hardware acceleration for performing heavy computations. Fred-Linux has been developed using the Zynq-7000 and Zynq-UltraScale+ by Xilinx as reference platforms, and it can be easily ported and extended on other platforms thanks to its modular design.

Keywords: Heterogeneous Computing, FPGA, DPR, Real-time, Linux

1. Introduction

In the last decade, computer platforms for embedded systems evolved towards heterogeneous architectures that comprise different processing elements and hardware accelerators. Such a development has been driven by the growing computational demand of modern cyber-physical systems such as autonomous vehicles and advanced robots. These systems need to acquire large amounts of data from multiple sensors and process them within precise timing constraints for performing the required control and monitoring actions. Such requirements result in the need to execute complex computing workloads such as machine learning, encryption, and advanced signal processing algorithms within precise timing constraints. Heterogeneous systems can meet this computational demand while retaining a high energy efficiency level by distributing the

computational workload among their different processing elements.

In particular, SoC-FPGA platforms, which comprise a system-on-chip device tightly coupled with FPGA hardware-programmable fabric, are especially suitable for real-time embedded systems due to the highly predictable nature of FPGA-based hardware acceleration. Compared to other types of hardware acceleration on heterogeneous platforms, like GPU co-processing, FPGA-based acceleration allows for precise control of the logic design. This feature is crucial for implementing predictable-by-design hardware accelerators and memory subsystems, which can provide high time predictability. As such, FPGA-based acceleration is attractive in several safety-critical domains.

Modern SoC-FPGA platforms support dynamic partial reconfiguration (DPR), which allows reconfiguring a portion of the FPGA fabric while the remaining logic cells continue to operate without interruption. By leveraging DPR, multiple hardware accelerators can share the same fabric resources in time-multiplexing. This capability is particularly interesting when considering that many real-time applications consist of periodic or sporadic activities. Hence, employing statically allocated hardware modules for accelerating recurrent software activities can result in an

*This work has been supported by the EU H2020 project AMPERE under the grant agreement no. 871669.

*Corresponding author

Email addresses: marco.pagani@santannapisa.it (Marco Pagani), alessandro.biondi@santannapisa.it (Alessandro Biondi), mauro.marinoni@santannapisa.it (Mauro Marinoni), giuseppe.lipari@univ-lille.fr (Giuseppe Lipari), giorgio.buttazzo@santannapisa.it (Giorgio Buttazzo)

underutilization of the FPGA fabric since logic resources may remain idle for a considerable amount of time. On the contrary, by leveraging DPR, it is possible to dynamically deploy the required hardware accelerators only when software activities actually need them.

However, although FPGA-based hardware acceleration offers numerous advantages for real-time systems, it also presents new challenges. For instance, scheduling concurrent acceleration requests while guaranteeing predictable delays is not trivial, especially when considering that the FPGA reconfiguration interface can reconfigure at most one portion of fabric at a time. The FRED framework [1] has been proposed as a solution to predictably handle FPGA-based hardware accelerators under DPR. FRED is based on a scheduling infrastructure designed to guarantee bounded response times for software activities that make use of hardware acceleration. The FRED scheduling infrastructure has been built upon a platform model that matches modern SoC-FPGA platforms.

1.1. Contribution

This paper presents a novel version of Fred-Linux, the reference implementation of the FRED framework for GNU/Linux originally proposed in [2]. Fred-Linux has been used to accelerate various kinds of workloads, including computer vision, matrix algebra, and inference of neural networks [3] implemented with the popular FINN framework [4]. The new version presented in this work provides a rich set of improvements and new features over the previous version of [2]. First of all, the central software component of Fred-Linux has been entirely redesigned from scratch in a more modular fashion to better support future extensions of the FRED framework. Second, new important features such as support asynchronous acceleration requests and watchdog timers for detecting Hw-tasks stalls are introduced. Third, on the kernel side, device reconfiguration is now managed by a new low-level driver based on the Linux vendor-independent FPGA manager framework. With this new reconfiguration driver, Fred-Linux now fully supports both the Zynq-7000 and Zynq UltraScale+ platforms by Xilinx. On the developer side, the client API has been significantly extended to be more flexible. A Python API has been also introduced to support the popular PYNQ framework [5]. Finally, the new implementation has been tested with a realistic case-study application using standard components such as Qt, OpenCV, and Video for Linux (V4L). The evaluation results show that real-world applications can practically benefit from Fred-Linux, being capable of leveraging time-predictable FPGA-based hardware acceleration in a feature-rich operating system environment such as GNU/Linux with low overhead. Fred-Linux is distributed under the GNU General Public License. The source code and the documentation are freely available online [6].

2. Related Work

Contemporary SoC-FPGA heterogeneous platforms allow leveraging FPGA-based hardware acceleration while DPR allows virtualizing the FPGA resources in the time domain. Nevertheless, the complexity of HW/SW co-scheduling and the non-negligible reconfiguration still poses considerable challenges. In particular, real-time applications can benefit from DPR, but mechanisms and analyses are required to enforce and guarantee timing constraints. However, timing analysis for such systems is still an open research topic since many proposed approaches focus on specific elements or apply oversimplified assumptions not matching real platforms' capabilities.

Many research effort has been spent on the reconfiguration interface, which is a crucial component enabling DPR. Dittmann and Frank [7] approached the issue of scheduling reconfiguration requests on a single shared non-preemptive reconfiguration interface, such as the Xilinx ICAP port. Duhem et al. [8] proposed a fast ICAP controller to reduce the overhead, using high-speed configuration and easy-to-use readback capabilities, coupled with a corresponding model for the reconfiguration overhead estimation. It enhances the performance by exploiting DMA, ICAP overclocking, bitstream pre-load into the controller, and bitstream compression. An analysis of the challenges of runtime reconfiguration in real-time systems was presented by Damschen et al. [9], which shows the massive impact of conflicts accessing a shared main memory during reconfiguration. They applied these results in the development of a runtime reconfiguration support under WCET guarantees. Valente et al. [10] proposed a detailed characterization of the components involved in the reconfiguration in order to provide a worst-case bound on the cost. This result allowed them to define the profitability of the DPR reconfiguration in real-time systems. The advantages of DPR-based solutions in terms of HW-tasks execution predictability and speed-up have been investigated by Pezzarossa et al. [11] to evaluate the improvements over a pure software solution and the trade-off between the use of multiple specialized accelerators combined with DPR instead of the use of a more general accelerator, both in terms of performances and memory footprint. Several ways to integrate DPR capabilities in real-time applications have been explored in order to mitigate the reconfiguration costs and reduce the related pessimism. Goossens et al. [12] addressed the problem of multi-mode systems, analyzing the impact of mode changes for FPGA-accelerated application, proposing a mode change protocol and test for the verification of the timing constraints.

Handling DPR at the task level is even more challenging since the reconfiguration time deeply affects the response times. Danne and Platzner [13] proposed a model requiring oversimplistic assumptions, such as preemptability of HW-tasks, negligible reconfiguration time, and no allocation constraints. The model proposed by Saha et al. [14] also analyzes preemptable HW-tasks on homogeneous partitions

with FPGA reallocation at each HW-task termination, inducing an area waste and increasing the pessimism in the analysis. Biondi et al. [1] proposed the FRED framework, a predictable infrastructure to supporting DPR of hardware accelerators. It can be implemented on existing platforms being based on a more detailed platform model. Saha et al. [15] proposed a co-scheduling framework for the execution of both periodic and aperiodic real-time tasks on DPR platforms that minimizes aperiodic tasks rejection while guaranteeing the feasibility of the periodic tasks.

Fruitfully exploiting DPR features on FPGA-based platforms requires robust and user-friendly support from the operating system perspective. The ARTICo3 framework proposed by Rodriguez et al. [16] provides an automated toolchain and runtime support to create and execute reconfigurable systems without additional user intervention transparently. Moreover, it allows adapting resources to tune the trade-off among computing performance, energy consumption, and fault tolerance.

However, predictable mechanisms and algorithms are mandatory when considering real-time applications to allow the definition of safe and tight bound. Lübbers and Platzner [17] presented ReconOS, an operating system that extends the traditional multi-threading programming model to HW-tasks running on a reconfigurable FPGA. The initial version addressed fully reconfigurable FPGAs and was later extended to include support for partial reconfiguration [18]. Interactions among threads are managed by common POSIX-like abstractions (e.g., semaphores, shared memory). R3TOS is an operating system proposed by Iturbe et al. [19] supporting the dynamic allocation of HW-tasks on an FPGA without the need for a preconfigured partitioning and static interconnections. The authors introduce a module, called HWuK, responsible for scheduling HW-tasks, performing their allocation, and managing the reconfiguration. A drawback of R3TOS is its intrinsic dependency on the reconfiguration interface (already a bottleneck for reconfiguration activities), which is further loaded for data communications. Pagani et al. [20] proposed a prototype implementation of the FRED framework over the FreeRTOS [21] kernel to show the applicability of the approach.

The Linux community has also shown interest in leveraging FPGA fabric. Brodersen proposed BORPH [22], which extends the Linux kernel to allow co-scheduling of SW-tasks and HW-tasks. However, the project is discontinued and does not consider modern platforms. More recently, the Linux community developed the FPGA Manager, a subsystem that allows reprogramming FPGAs in a vendor-agnostic fashion [23]. While the FPGA Manager is a major improvement over the previous vendor-specific interfaces, its responsibilities are understandably limited to low-level FPGA management. Hence, it does not include a scheduling infrastructure nor a standard interface for FPGA modules.

This paper presents a novel implementation of the FRED framework for leveraging FPGA acceleration, ad-

ressing some limitations present in previous work. The proposed solution has been designed to efficiently use the available resources by including zero-copy data transfer mechanisms, support for asynchronous acceleration requests, and support for watchdog timers for stall detection. Furthermore, it does not impose any specific paradigm for the HW-task design (e.g., stream processing, data flow) and it is easily extendable to different platforms (currently supports the Zynq-7000 and the Zynq UltraScale+ by Xilinx). The proposed implementation is integrated with the standard Linux FPGA manager to enhance its usability in real-world systems.

3. Background: the FRED framework

This section briefly reviews the FRED framework proposed in [1], on which this work is based. FRED is a framework designed for supporting predictable hardware acceleration of real-time applications using DPR. It allows hosting in time-sharing a larger number of hardware accelerators with respect to the number of accelerators that can be statically allocated on the physical fabric. The FRED framework is based on a general platform model conceived to account for several real-world technological constraints present on modern SoC-FPGA platforms.

3.1. Platform model

FRED considers heterogeneous computing platforms consisting of one or more general-purpose processors coupled with a dynamically-reconfigurable FPGA. Both sides of the platforms, the general-purpose processors and the FPGA, have access to a shared memory \mathcal{M} , as illustrated in Figure 1. The FPGA fabric contains b logic blocks, which represents an abstraction of the specific physical cells available on the fabric. The FPGA fabric is statically partitioned into a set of n_P partitions $P = \{P_1, \dots, P_{n_P}\}$, where each partition P_k is composed of b_k logic blocks, with $\sum_{k=1}^{n_P} b_k \leq b$. Logic blocks are not shared among partitions. Each partition P_k is further split into n_k^S slots of b_k^S logic blocks, such that $\forall P_k \in P, n_k^S \cdot b_k^S \leq b_k$. Logic blocks are not shared among the slots.

The FRED naming convention gets its inspiration from the terminology used in the real-time community for partitioned scheduling. Unfortunately, this established naming convention conflicts with Xilinx’s terminology. For this reason, please remember that FRED’s *slots* correspond to *reconfigurable partitions* (floorplanned using Pblocks) in Xilinx terminology. In contrast, FRED’s partitions are just logical containers (i.e., groups) of slots.

Partitions have been introduced since HW-tasks tend to have heterogeneous resource requirements depending on the specific function implemented. However, due to technological constraints, when a HW-task is implemented in a slot (Pblock), it occupies the entire area, regardless of the number of resources actually being used. Nevertheless, a FRED design may use several HW-tasks with similar

resource requirements. Hence, it is convenient to define the concept of partition as a group of slots having the same size to improve resource utilization. Furthermore, a partition can include multiple slots as the designer may want to run multiple HW-tasks parallel.

The geometrical placement of the slots belonging to the n_P partitions within the physical FPGA fabric, i.e., the problem of floorplanning, is beyond the scope of this paper. Interesting solutions were proposed for addressing this problem. For instance, Seyoum et al. proposed an automated floorplanner based on optimization via Mixed-Integer Linear Programming (MILP) [24].

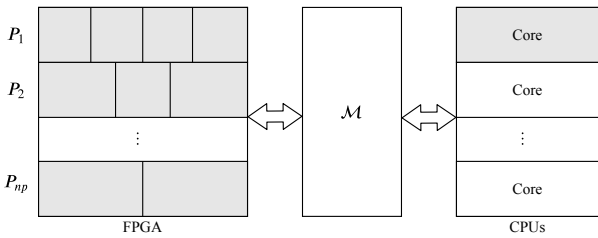


Figure 1: FRED platform model.

3.2. Application model

FRED considers two kinds of computational activities: **(i) software tasks** (SW-tasks), and **(ii) hardware tasks** (HW-tasks). SW-tasks are conventional software activities running on one of the general-purpose processors, while HW-tasks are instances of hardware accelerators designed to be configured and executed on the FPGA fabric. SW-tasks can speedup parts of their computation by requesting the execution of HW-tasks on the FPGA fabric. More formally, a FRED application is composed of two sets of activities; (i) a set of n_S SW-tasks $\Gamma^S = \{\tau_1, \dots, \tau_{n_S}\}$, and (ii) a set of n_H HW-tasks $\Gamma^H = \{\tau_1^H, \dots, \tau_{n_H}^H\}$.

Communication between SW-tasks and HW-tasks is supported by allowing HW-tasks to directly access the shared memory \mathcal{M} . The problem of modeling memory contention between SW-tasks and HW-tasks is beyond the scope of this paper (the interested reader can refer to [25, 26]). However, for the scope of this work, the WCET of the HW-tasks can be empirically determined by executing all HW-tasks concurrently using a dedicated test bench to simulate the maximum contention scenario. Furthermore, it is worth noting that the contention depends entirely on the hardware platforms since HW-tasks are implemented using dedicated logic resources and typically produce regular transaction patterns that can be well characterized both in logic simulation and execution. Please refer to the work of Restuccia et al. for further details about memory contention for FPGA accelerators [27].

3.3. Hardware task and reconfiguration interface model

Each HW-task $\tau_i^H \in \Gamma^H$ is an instance of a hardware accelerator requiring b_i^H logic blocks and having a *worst-case execution time* (WCET) C_i^H . A HW-task can execute

only after being configured on one of the slots available on the FPGA fabric. Once started, the execution of HW-tasks cannot be preempted.

FRED assumes that the heterogeneous platform is equipped with an FPGA reconfiguration interface (FRI), which can dynamically reconfigure a slot at run-time to deploy HW-tasks. Each slot can accommodate at most one HW-task [28, 29]. The FRI has been modeled to match the capabilities and limitations of real-world platforms (such as [30, 28]). Hence, it is assumed that:

1. the FRI can reconfigure a slot without affecting the execution of the HW-tasks currently running in other slots;
2. a negligible amount of processor cycles are used for reconfiguring a slot, i.e., the FRI is fed by a paired direct memory access (DMA) engine. Hence, the processor does not busy wait until reconfiguration completes;
3. the FRI can reconfigure at most one slot at a time.

To reconfigure 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^H of logic blocks required by τ_i^H . Each HW-task τ_i^H can be reconfigured 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 $b_i^H \leq b_k^S$.

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

3.4. Software task model

Each SW-task $\tau_i \in \Gamma^S$ can make use of the HW-tasks in Γ^H to accelerate its computations and is subject to timing constraints. Each SW-task τ_i uses a set $\mathcal{H}(\tau_i) \subseteq \Gamma^H$ of m_i HW-tasks and alternates the execution of $m_i + 1$ software sub-tasks 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

$$\tau_i := \langle \tau_{i,1}, \tau_a^H, \tau_{i,2}, \tau_b^H, \dots, \tau_{i,m_i+1} \rangle,$$

where $\{\tau_a^H, \tau_b^H, \dots\} \in \mathcal{H}(\tau_i)$ and $\tau_{i,j}$ is the j -th sub-task of τ_i . Two acceleration schemes are supported, namely *synchronous* and *asynchronous*. Under synchronous acceleration, 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. Conversely, when asynchronous acceleration is used, the execution a SW-task can continue after issuing the execution of a HW-task. A synchronization method is

```

1 sw_task( $\tau_i$ )
2 {
3   <...>
4   <prepare input data for  $\tau_a^H$ >
5   fred_accel( $\tau_a^H$ );
6   <retrieve output data from  $\tau_a^H$ >
7   <...>
8   <prepare input data for  $\tau_b^H$ >
9   fred_accel( $\tau_b^H$ );
10  <retrieve output data from  $\tau_b^H$ >
11  <...>
12 }

```

Listing 1: Pseudo-code of the implementation skeleton of a SW-task.

then later invoked to wait for the completion of the execution of the HW-task, possibly self-suspending the calling SW-task as under synchronous acceleration if the HW-task is not yet completed. At most, one acceleration request at a time can be pending for each SW-task in both *synchronous* and *asynchronous* acceleration schemes. Please note that while a single SW-task can have at most one acceleration request can be pending at a time, multiple SW-tasks can concurrently issue multiple acceleration requests.

Each SW-task τ_i has a cumulative WCET C_i and is periodically (or sporadically) released with a period (or minimum inter-arrival time) of T_i units of time, hence generating an infinite sequence of execution instances (denoted as jobs). Finally each SW-task τ_i 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 $\bigcap_{\tau_i \in \Gamma^S} \mathcal{H}(\tau_i) = \emptyset$. Listing 1 reports a pseudo-code that is representative of the body of a SW-task τ_i using $m_i = 2$ HW-tasks in the set $\mathcal{H}(\tau_i) = \{\tau_a^H, \tau_b^H\}$ with synchronous acceleration. The statement $\langle \dots \rangle$ represents some code block containing a sequence of instructions that will be executed on the general-purpose processor.

The SW-task illustrated in Listing 1 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. `fred_accel(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 at line 6 and prepares the input data for τ_b^H at line 8. A sample schedule of a SW-task that uses two HW-tasks is illustrated in Figure 2.

3.5. Scheduling Infrastructure

The FRED framework comes with a scheduling mechanism to handle the contention of the FRI and the FPGA slots. The scheduling mechanism is based on a multi-level queueing structure as illustrated in Figure 3, which includes (i) n_p *partition queues* (one for each partition), needed to schedule the requests for HW-tasks with affinity to the

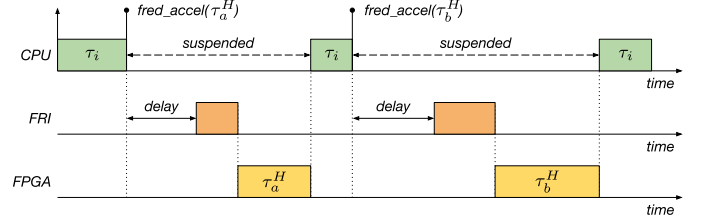


Figure 2: Sample schedule of a SW-task using two HW-tasks. The up-arrow denote the release of the SW-task τ_i .

same partition, and (ii) a *FRI queue* to schedule the re-configuration requests. The partition queues are ordered according to the first-in-first-out (FIFO) policy. Each time a SW-task issues an execution request \mathcal{R} for a HW-task, \mathcal{R} is assigned a *ticket* marked with the current absolute time. Then, \mathcal{R} is inserted into its corresponding partition queue (depending on the affinity of the HW-task). The partition queues enqueue a request as long as there are no free slots into the corresponding partition. The FRI queue is fed by the partition queues and is ordered by *increasing ticket time*. This mechanism guarantees that acceleration requests are served with predictable worst-case delays, which have also been analytically bounded by response-time analysis [1].

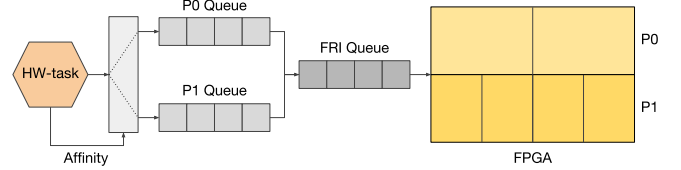


Figure 3: Multi-level queueing structure for scheduling HW-tasks. The FPGA area is divided into two partitions P_0 and P_1 .

3.6. Response-time analysis

For the sake of completeness, the response-time analysis for the FRED framework (from [1]) is reported next. Non-preemptive management of the FRI, as used by Fred-Linux, is considered.

Theorem 1 (from [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 non-preemptive management of the FRI, the maximum delay Δ_a incurred by \mathcal{R}_a is upper-bounded by*

$$\overline{\Delta}_a^{NP} = \overline{\Delta}_a^P + NH_k^{max} \times r_k^{max} \quad (1)$$

where

$$\overline{\Delta}_a^P = \sum_{\tau_j \neq \tau_i} \max_{\tau_b^H \in \mathcal{H}(\tau_j)} \{ \Delta_b^{slot} + r_b \}, \quad (2)$$

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

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

Please refer to [1] for a precise formalization of the scheduling rules and a complete description of the response-time analysis.

4. Platform support

This section describes a system design for supporting the FRED framework on top of Xilinx’s SoC-FPGAs platforms, such as Zynq-7000 and Zynq UltraScale+, which have been chosen as the reference platforms for FRED due to their popularity. Depending on the specific generation, they include a cluster of ARM Cortex-A v7 or v8 processors tightly coupled with a reconfigurable FPGA fabric. The internal structure of Xilinx SoC-FPGAs is divided into two main functional blocks: (i) the *processing system* (PS) block and, (ii) the *programmable logic* (PL) block [31]. The PS block includes the cluster of Cortex-A processors, a set of memory controllers for driving external memories, a small amount of on-chip RAM, and various I/O peripherals. On recent platforms, such as the Zynq UltraScale+, the PS also includes a cluster of Cortex-R processors and two MicroBlaze-based processing units named platform management unit (PMU) and configuration security unit (CSU). The PMU is in charge of platform and power management, while the CSU monitors system integrity and safety. The PL block includes a reconfigurable FPGA fabric consisting of a bidimensional array of programmable logic resources. The specific quantity and type of logic resources depend on the specific generation and SoC model. The units included in the PS side, i.e., ARM cores, memory controllers, and peripherals, are interconnected through AMBA AXI memory-mapped interfaces. The AMBA AXI standard allows simultaneous, bi-directional data exchange between master and slave interfaces. The master interface initiates the transactions, and the slave interface responds to the requests. Multiple interfaces can be connected together through an interconnect block. The interconnect arbitrates the transactions and performs protocol and data-width conversions. The same AXI infrastructure is exported to the PL through a set of memory-mapped AXI interfaces exported. These AXI interfaces can be used to extend the system by connecting additional logic modules deployed on the PL.

4.1. System support design

The FRED support design provides the foundations for the software support, enabling the interleaving of dynamically-reconfigured HW-tasks on the PL fabric. Figure 4 provides a schematic representation of the design. The PL area is partitioned into two main regions: (i) a *static* region, and (ii) a *reconfigurable* region for hosting hardware accelerators. The static region contains the AXI interconnection infrastructure, namely a set of AXI Interconnects (discussed in Section 4.1.4), and may host other support modules in an application-dependent fashion. The reconfigurable region is sub-partitioned into a set of slots that are logically grouped

into partitions as detailed in Section 3. As discussed in [1], a slotted approach is more suitable for real-time systems since no allocation and defragmentation overhead is introduced. Moreover, Xilinx tools natively support static partitioning (using Pblocks [28]), which allows constraining the implementation of a logic module to a geometrical region of the FPGA. Therefore, a FRED design flow can be implemented using commercial design tools without relying on third-party experimental solutions, which would instead be required for a slotless approach.

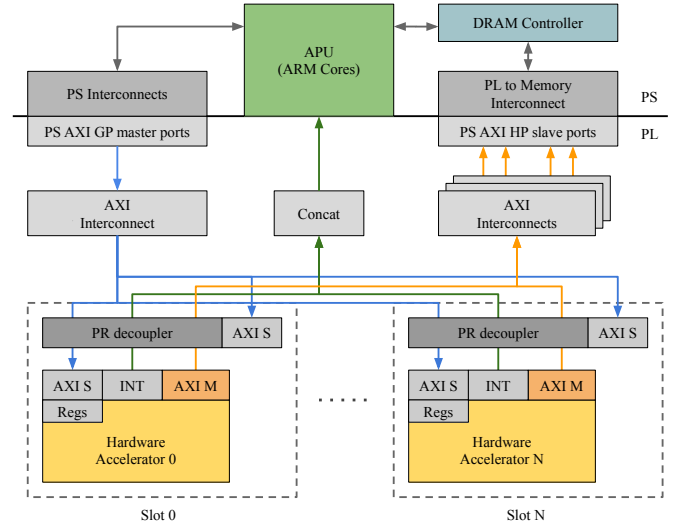


Figure 4: Fred-Linux support design for Xilinx platforms.

According to shared-memory communication paradigm of FRED discussed in Section 3, each HW-task must be able to autonomously access memory regions that are also available to the processors. Xilinx’s SoC-FPGAs provide three alternatives for implementing such memory regions: (i) using the internal on-chip memory; (ii) using PL resources to build custom memories on the PL (using BRAM logic blocks); or (iii) using the main (off-chip) DRAM memory. Alternative (i) is not viable since the on-chip memory is limited to 256 KB on both Zynq UltraScale+ and Zynq-7000 [31, 32], and hence may be unsuitable for supporting shared-memory communication for systems with multiple HW-tasks. Alternative (ii) may determine a loss of available FPGA resources since implementing large memory buffers on PL consumes a significant amount of BRAM logic blocks. Conversely, alternative (iii) allows taking advantage of the high-performance AXI ports (HP ports) that grant direct access to the DRAM controller from the PL. Fred-Linux follows the latter approach, implementing the shared-memory communication paradigm using the off-chip DRAM memory.

In FRED, each HW-task has affinity to a partition and can be configured and executed in any slot belonging to that partition. This requirement implies that each slot must be able to host any HW-tasks associated with his partition. Given the technological constraints of Xilinx’s SoC-FPGA platforms [28], this requirement can be fulfilled

by defining a *common interface* that must implement by each HW-task deployed on the system, which is presented next.

4.1.1. Common interface

The proposed common interface for HW-tasks consists of (i) at least one AXI master interface, (ii) an AXI-Lite slave interface exporting a predefined set of control registers and eight data registers, and (iii) an interrupt signal to send notifications. The AXI master interfaces (denoted as AXI M in Figure 4) allow HW-tasks to access the main memory through the PS DRAM controller, hence implementing the shared-memory paradigm. In this way, HW-tasks can autonomously retrieve the data they need to process without the need for intervention by the processors. It is worth noting that *bus mastering* is also crucial for supporting high-performance hardware accelerators that need to process large amounts of data.

The AXI-Lite slave interface (denoted as AXI S in Figure 4) allows mapping the control and data register of HW-tasks into the address space seen by the processors so that they can be controlled by Fred-Linux. All HW-tasks must implement the same set of control and data registers map to allow the usage of a common software driver. The eight data registers are used to exchange memory pointers whose meaning depends on the specific function implemented by the HW-task. Finally, the interrupt signal (denoted as INT in Figure 4) is meant to be connected to the interrupt controller for notifying the completion of the HW-task to the processors.

Fred-Linux has been designed with high-level synthesis support in mind to simplify the implementation of computationally-intensive functions as FPGA-based hardware accelerators using the popular Vivado HLS tool. The system designer can generate Fred-Linux-compliant HW-tasks by wrapping the C or C++ code of the functions to be accelerated into the common top-level wrapper reported in Listing 2. Vivado HLS will automatically generate the standard interface logic thanks to its interface synthesis capabilities. In addition to HLS, it is also possible to develop HW-tasks directly using hardware description languages such as VHDL or Verilog for achieving higher performance. In this case, a VHDL stub is provided by Fred-Linux, while the Xilinx Vivado suite already provides HDL code stubs for implementing the AXI master and AXI-Lite slave interfaces.

4.1.2. Dynamic partial reconfiguration support

In Xilinx’s SoC-FPGAs, the software running on the PS can fully or partially reconfigure the PL fabric using the *processor configuration access port* (PCAP). The PCAP is fed by a DMA engine that can be programmed to transfer a bitstream from the system memory to the PL configuration memory. On recent Zynq UltraScale+ platforms, the configuration DMA is included in the CSU unit, while on the Zynq-7000 series, it is included in the device configuration interface (DevC) subsystem. Compared to other

Listing 2: Vivado HLS code for implementing HW-tasks.

```

1 void slot_i(args_t *id, args_t args[ARGS_SIZE], volatile
   data_t *mem_in, volatile data_t *mem_out)
2 {
3   // AXI Lite control bus
4   #pragma HLS INTERFACE s_axilite port=return bundle=
   ctrl_bus
5   #pragma HLS INTERFACE s_axilite port=id bundle=ctrl_bus
6   #pragma HLS INTERFACE s_axilite port=args bundle=ctrl_bus
7
8   // AXI Master memory ports
9   #pragma HLS INTERFACE m_axi port=mem_in offset=slave
   bundle=mem_bus
10  #pragma HLS INTERFACE s_axilite port=mem_in bundle=
   ctrl_bus
11  #pragma HLS INTERFACE m_axi port=mem_out offset=slave
   bundle=mem_bus
12  #pragma HLS INTERFACE s_axilite port=mem_out bundle=
   ctrl_bus
13
14  fred_hwacc_body(id, args, mem_in, mem_out);
15 }

```

configuration paths, such as the *internal processor configuration access port* (ICAP), the PCAP is driven by control logic included on the PS side of the device. Hence, it does not consume additional PL fabric resources to be instantiated. The Xilinx’s standard toolchain does not support bitstreams relocation [28], i.e., the same bitstream cannot be used to program the same HW-tasks in different slots. This limitation can be overcome by synthesizing a different bitstream for each slot of the partition to which the corresponding HW-task belongs. More formally, for each HW-task $\tau_i^H \in \Gamma^H$, belonging to partition P_k , there are n_k^S bitstreams, one for each slot. This approach trades a higher memory consumption with the advantage of not requiring third-party bitstreams relocation tools, which the vendors do not guarantee. However, we believe that this does not constitute a limitation since partial bitstreams images are typically in the order of a few megabytes.

4.1.3. Slot decouplers

During the FPGA reconfiguration process, the behavior of the area under reconfiguration is undefined since its logic resources may be in an inconsistent state. In particular, logic resources may produce transient signals while being programmed. These signals can cause troublesome spurious transactions in other modules, such as the AXI interconnects or the system interrupt controller. To protect the system for these events, each reconfigurable slot is protected by a partial reconfiguration decoupler (denoted as PR decoupler in Figure 4), a Xilinx’s library IP that binds the wires of the slot interface to safe logic values during the reconfiguration process [28]. Fred-Linux controls each decoupler through a single control register mapped into the system address space through an AXI-Lite slave interface.

4.1.4. Interconnections

In the FRED platform model, all HW-tasks employ bus mastering techniques to access the system memory and share data with SW-tasks running on the ARM cores. Hence, bus and memory access represent a crucial contention point. The problem of controlling bus and memory contention in a predictable fashion is beyond the scope of this paper — the research community proposed different techniques for improving memory and bus predictability, which can be integrated into the FRED system support system design (the interested reader can refer to [33, 34, 35]).

Within the scope of Fred-Linux, the interconnection infrastructure of the support design has been designed to evenly distribute the bandwidth supplied by the HP ports to all HW-tasks. If the number of HP ports available to the PL are enough to connect the master interface of each slot to a dedicated port, then this scheme is followed. Otherwise, the master interfaces of the slots are connected to a slot-level *N-to-1* interconnect block [36, 37] to obtain a single AXI master interface, which in turn can be connected to one of the available HP ports through a port-level interconnect. It is worth noting that the support design only depends on the total number of slots and not on the number of HW-tasks.

4.2. Floorplanning

As mentioned in Section 3.1, the problem of floorplanning (i.e., the geometrical placement of the slots on the FPGA fabric) is beyond the scope of this paper. However, it is worth mentioning that simple designs, containing few slots, can be floorplanned manually by the system designer. Larger designs can be automatically floorplanned using FLORA [24], an automated floorplanner based on optimization via Mixed-Integer Linear Programming (MILP). Moreover, the entire development flow of a Fred-Linux application can be automatized (including the generation of the static part) using DART [38]. These tools enrich the ecosystem around the runtime support presented in this paper.

5. Linux support

This section describes the architecture of the Fred-Linux software support, which is built on top of the hardware design presented in previous section. The software support has been designed in a modular fashion, relying as much as possible on user-space components for improving maintainability, portability, and extendability. Its internal architecture is shown in Figure 5. The central component of the software support is a user-space server process, named *fred-server*, which is in charge of managing acceleration requests from SW-tasks.

Periodic SW-tasks can be implemented as regular Linux processes or threads using the POSIX-compliant SW-task body presented in Listing 3. The SW-task body repeatedly

(i) performs computations by invoking one or more HW-tasks and (ii) makes use of POSIX’s `clock_nanosleep()` function for suspending and waiting for the next activation. Since Linux makes use of virtual memory, each SW-task process can access only its own private virtualized address space. On the other hand, HW-tasks are custom hardware components directly accessing the physical address space where the DRAM memory is mapped through the AXI bus. Implementing the shared-memory paradigm of FRED (described in Section 3) requires the development of an efficient mechanism to share data between the virtual and the physical memory domains. Recent SoC-FPGA platforms like the Zynq UltraScale+ include an IOMMU that allows AXI masters deployed in the PL to have a virtualized view of the system memory. However, older platforms like the Zynq-7000 do not include an IOMMU. Hence, HW-tasks are limited to a physical view of the system memory. To provide a uniform yet efficient implementation of the communication mechanism, Fred-Linux relies on a zero-copy design, using coherent memory buffers as communication channels between SW-tasks and HW-tasks. Zero-copy data movement is achieved using Linux’s memory management which implements paged virtual memory. This memory management technique allows mapping the memory pages that constitute the shared memory buffers into the virtual address space of a process by configuring its page table. This operation is performed, during the system initialization stage, using the Linux’s `dma_common_mmap()` function. In this way, a SW-task and a HW-task can share the same memory buffer without any additional overhead.

Listing 3: Pseudo-code stub for a SW-task.

```
1 void sw_task_stub(void *args)
2 {
3     struct timespec ts;
4     int period_ms = <task_period>;
5
6     /* Get current time */
7     clock_gettime(CLOCK_MONOTONIC, &ts);
8     /* Set next activation */
9     time_add_ms(&ts, period_ms);
10
11     while (true) {
12         /* SW-task body: */
13         { <First software chunk> }
14         { <Call HW-task> }
15         { <Second software chunk> }
16         { <Call HW-task> }
17         { <Third software chunk> }
18
19         /* Sleep until next activation */
20         clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &ts,
21                        NULL);
22         /* Set next activation */
23         time_add_ms(&ts, period_ms);
24     }
25 }
```

5.1. Kernel-space components

Fred-server leverages a custom kernel-level support for performing the low-level operations required to control the hardware components of the system support design. It consists of (i) a custom kernel module for allocating the memory buffers employed to share data between SW- and HW-tasks, and (ii) a semi-custom low-level FPGA driver for managing device reconfiguration with the FPGA manager framework. The latter is device-specific: Fred-Linux comes with two different versions for the Zynq-7000 device and the more modern Zynq UltraScale+ platforms. These components are discussed next in details. Fred-server also relies on the UIO framework for managing HW-tasks, i.e., accessing control and data registers, and observing the interrupt lines.

5.1.1. Buffers allocator module

The shared-memory infrastructure described in Section 3 has been implemented through a set of memory buffers allocated by a custom kernel allocator module. The allocator module uses the Linux DMA layer to allocate physically-contiguous, uncached memory buffers to exchange data between SW-tasks and HW-tasks. When loaded, the allocator module creates a new character device named `fred_buffctl`, which is used by fred-server during the initialization phase for requesting the allocation of memory buffers. Each allocation request is performed using the `ioctl` syscall, passing the required buffer size as an argument. On the kernel side, when the driver receives an allocation request, it creates a new character device named `fred_buffN` (where N refers to the buffer identifier that is assigned by the module) and allocates a new contiguous memory buffer, associated with the device, using the `dma_alloc_coherent()` function of the Linux DMA layer. The character device is the means by which the buffer is accessible from user-space.

Once the buffer device has been created, it can be accessed by a SW-task through memory mapping using the Linux `mmap()` syscall. When a SW-task calls (from user-space) the `mmap()` on a buffer character device, the associated memory buffer will be mapped into its virtual address space. Inside the allocator module (on the kernel side), the mapping is performed using the `dma_common_mmap()` function of the Linux DMA layer. Once the buffer is mapped into the SW-task's virtual address space, it can be accessed by the SW-task to read and write data without any system overhead. Since the buffer is uncached, no flush and invalidate operations are required on the cache. On the other side, a HW-task can access the same buffer through a physical memory address. The buffers' physical addresses are passed to the HW-tasks by fred-server using the set of data registers of their interface (see Section 4.1.1). In this way, SW-tasks and HW-tasks can efficiently share data without any additional copy operation or operating system overhead. It is worth observing that, with this design, the SW-tasks never deal directly with memory management

operations. From the programmer's perspective, the process of mapping these buffers, likewise all other interactions with fred-server, is assisted by the client support library described in Section 6. During the system shutdown phase, fred-server releases the buffer devices created during the initialization phase using the `ioctl` syscall again on the `fred_buffctl` control device.

5.1.2. FPGA driver

The FPGA Manager is a component of the Linux's FPGA subsystem that allows reprogramming FPGAs under Linux in a vendor-agnostic fashion [23]. The FPGA Manager has multilayered architecture consisting of an upper layer and a lower layer. The upper layer presents to the programmer a vendor-agnostic API that hides all the platform-specific details. Conversely, the lower layer consists of an FPGA driver, which provides a concrete implementation of the operations invoked by the upper layer for reconfiguring the FPGA. The FPGA driver is in charge of low-level operations such as modifying the bitstream and programming the configuration DMA, which are necessary for reconfiguring the FPGA. Since these operations are inherently platform-dependent, Xilinx provides a specific FPGA driver for the Zynq-7000 and another for the Zynq UltraScale+.

The high-level interface exported by the FPGA Manager includes a set of operations for loading a bitstream using Linux's firmware layer or using the In-kernel API. However, the high-level interface lacks support for I/O multiplexing, i.e., polling over a set of file descriptors using the `select()`, `poll()`, or the Linux-specific `epoll()` system call. Such a capability is required by fred-server being an event-driven application that leverages I/O multiplexing for monitoring multiple events sources. Furthermore, the current version (2019.2) of the Zynq UltraScale+ FPGA driver does not support direct bitstream copy from a contiguous buffer. Instead, for each reconfiguration request, the driver internally allocates a contiguous uncached memory buffer using the `dma_alloc_coherent()` function of the Linux DMA layer. Once the buffer has been allocated, the driver copies the bitstream image to the buffer and then calls the `fpga_load` method of the embedded energy management interface (EEMI) to begin the reconfiguration process. Then, the request is handled by the ARM trusted firmware (ATF) and dispatched to the PMU through an inter-processor interrupt (IPI). The PMU handles the request and configures the CSU's DMA using the `xilfpga` software library [39].

The overhead introduced by the copy operation in the current release of the Zynq UltraScale+ FPGA driver and the lack of support for I/O multiplexing by the FPGA Manager make these components unsuitable for the intensive usage of partial reconfiguration as required by FRED. To overcome these limitations, the low-level Zynq UltraScale+ FPGA driver has been extended for supporting both I/O multiplexing and the direct load of contiguous bitstream images. These new functionalities are exported

to the user through a set of four sysfs attributes. The first two attributes are used for passing a reference to a contiguous bitstream image to the driver. The third attribute is used for starting the reconfiguration. Once the reconfiguration is started, the fourth attribute can be monitored through POSIX standard I/O multiplexing methods, such as `select()` and `poll()` or the Linux-specific `epoll()`, to receive a notification when the reconfiguration is completed. Please note that the aforementioned extended FPGA driver is still compatible with the FPGA manager framework. Hence, the FPGA can still be reconfigured using the regular high-level interface exported by the FPGA Manager upper layer.

5.2. User-space components

Fred-server is the central user-space component of Fred-Linux. From an architectural perspective, fred-server is an event-driven application that handles service requests coming from multiple event sources like SW-tasks issuing acceleration requests, HW-tasks notifying their completion, and other hardware events like the completion of the FPGA reconfiguration process. From a functional perspective, fred-server interacts with the rest of the system by means of two main software interfaces, one dedicated to inter-process communications with SW-tasks and the other to communicate with Linux and the kernel support, as illustrated in Figure 5. The communication interface between fred-server and SW-tasks is implemented using UNIX domain sockets. In this way, SW-tasks are entirely decoupled from fred-server.

During the initialization phase, fred-server reads a set of files describing the system layout and the available HW-tasks. Then, according to such a system description, it initializes the support, using the allocator kernel module to instantiate the memory buffers used for both bitstreams and data sharing. After the initialization phase, the server opens a listening socket used by SW-tasks to establish a new connection. Once the connection is established, the SW-task can send requests to the server.

From a client programmer perspective, communication functions between SW-tasks and fred-server are encapsulated into a client support library (see Sec. 6) to ease the development process. It is worth noticing that SW-tasks never directly interact with the hardware, nor they are required to perform privileged operations. Indeed, fred-server mediates any interaction between client SW-tasks and the platform hardware.

5.2.1. Fred-server internals

Fred-server is written in standard C99 using POSIX and Linux APIs. Compared to the previous version from [2], the fred-server has been redesigned in a modular fashion according to the reactor design pattern. The reactor pattern is an event handling pattern used for implementing event-driven applications capable of serializing and dispatching service requests concurrently issued from multiple

event sources [40, 41]. In particular, the reactor pattern decouples the responsibility of receiving and demultiplexing events from the responsibility of actually handling events. This characteristic is particularly useful within FRED, since multiple software and hardware event sources like SW-tasks, HW-tasks, FPGA reconfiguration interface, etc. need to be handled in different ways. In this context, the reactor pattern allows implementing each event handler with a different class derived from a common abstract base class. This approach conforms to the *single responsibility* principle [42] since each handler class needs to change only if the handling logic of the corresponding event needs to change. Moreover, it also respects the *open-close* design principle [42] since new classes of events (e.g., handling IPC signals) can be managed by implementing a new handling class without the need of modifying existing handler classes. Please note that, although C99 does not provide native language support for object-oriented programming, there are established techniques for supporting the object-oriented programming paradigm in C99 [43, 44]. The internal architecture of the fred-server is illustrated in Figure 6, highlighting the most relevant interactions between its key internal components. In more detail, the fred-server is composed of the following components:

- The **Event_Handler** is the abstract base class defining the interface of an event handler object. It must be inherited by all concrete event handlers classes, which have the responsibility of serving a specific type of event. All event handler instances contain a handle component, which is tied, during the initialization of each instance, to an operating system object (i.e., a file description) identifying the actual event source.
- The **Reactor** abstract base class defines the interface for registering handlers and for running the event loop. Concrete implementations must provide methods for (i) registering new event handlers and (ii) implementing the event loop logic for cyclically waiting over the handles provided by the set of registered events handlers. When an event occurs (i.e., a handle becomes ready), the event loop serves the event by calling the event handling logic contained in the event handler object that owns the handle. Reactors rely on synchronous event demultiplexers provided by the operating system (e.g., `poll`, `select`, etc.) for waiting on the set of registered handles. Currently, Fred-Linux provides two concrete implementations of the reactor that must be used in a mutually exclusive way depending on the specific needs. The first implementation is based on the `poll` function of the POSIX standard. The second implementation is based on the more recent `epoll` mechanism provided by Linux. The main advantage of the `epoll` function is that its time complexity is constant (i.e., $\mathcal{O}(1)$) with respect to the number of monitored handles. On the contrary, the time complexity of the classic

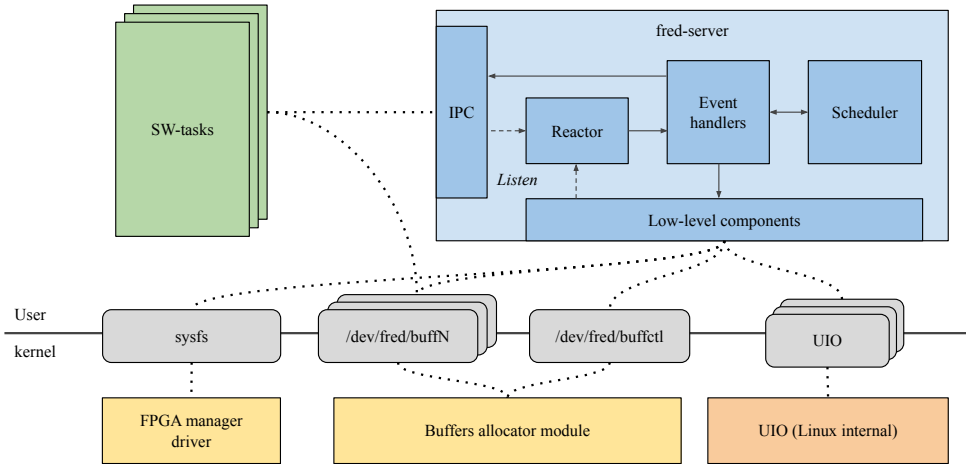


Figure 5: Overview of the fred-server.

`poll` function is linear with the number of handles. However, given the limited number of event sources present in a typical Fred-Linux design, and considering the limited amount of overhead introduced by the whole event handling logic, the performance gap between the two implementations is very marginal in practical cases.

- The `Software_Task_Listener` module is a concrete event handler that is in charge of registering SW-tasks during the initialization phase. This handler contains a listening socket handle. Whenever this handler receives a valid initialization request from a SW-task, it creates a new `SW_Task` object that handles the new connection socket and registers it to the reactor.
- The `Sw_Task_Client` module is a concrete handler representing an active SW-task and containing its connection socket handle. Each `Sw_Task_Client` is associated with a set of `Hw_Tasks` objects, representing the HW-tasks that the SW-task can call to accelerate its execution. Moreover, it owns a set of `Data_Bufferer` objects, which are coherent buffers objects used to exchange data with the associated HW-tasks.
- The `Slot` module is a concrete handler representing a “physical socket” for HW-tasks, i.e., a fixed portion of the FPGA area where a HW-task can be plugged in by means of partial reconfiguration. When initialized, the `Slot` starts in the `empty` state, meaning that no HW-task is actually configured into the physical slot. Once the `Slot` has been reserved to a SW-task, it goes into the `reserved` state where it is ready for reconfiguration. After reconfiguration, the `Slot` goes into the `ready` state, meaning that the contained HW-task is ready to execute. At the end of the HW-task execution, the `Slot` goes into an `idle` state waiting to be reconfigured. However, if the same HW-task needs to be executed again and the slot has not been used for another HW-task in the meanwhile, the reconfiguration process is skipped to save time. The `Slot` object controls HW-task configured in the corresponding slot and the related decoupler using, in turn, the `Slot_Drv` and `Decoupl_Drv` low-level components. These components allow decoupling the FRED control logic from the responsibility of performing the low-level control actions, which clearly depend on the specific platform. In particular, the `Slot_Drv` component exports the file handle owned by the `Slot` object that is used by the reactor to know that the contained HW-task completed its execution.
- The `Dev_Rcfg` module is a concrete handler that represents the reconfiguration device. It relies on a low-level component called `Rcfg_Drv` for controlling the specific reconfiguration engine of the platform. The handle exported by the `Rcfg_Drv` is used by the reactor to know when a reconfiguration is completed.
- The `Signal_Receiver` module is an optional concrete handler that can be used for handling inter-process communication signals using Linux’s `signalfd` function. If this handler is registered to the reactor, it allows receiving and handling standard signals like `SIGTERM` and others.
- The `Timer` module is a concrete handler that implements a watchdog timer for monitoring stalls and execution overruns of HW-tasks. Stalls and overruns are particularly dangerous as they can significantly jeopardize the whole system. For instance, if a HW-task never completes its execution, the slot in which it is configured remains unusable. When an overrun or a stall are detected, the interested HW-task is disabled and can no longer be requested by SW-tasks. Internally, the module includes a timer sub-component implemented using Linux’s `timerfd` API.
- The `Scheduler` module is a central component in charge of implementing the FRED scheduling policy

described in Section 3. All event handlers components notify the scheduler when an event occurs. In turn, the scheduler performs the required actions.

- The `Sys_Layout` module models the physical layout of the system. A Fred-Linux system is composed of a set of partitions, each of them containing a set of slots, and a set of HW-tasks. During the initialization process, the `Sys_Layout` module parses two configuration files that describe the system layout. The first file specifies the layout of the FPGA in terms of partitions and slots. The second file defines the available HW-tasks. According to the content of these files, the `Sys_Layout` modules initialize the system by instantiating all the components and registering to the reactor (i) all hardware-related handlers such as `Dev_Rcfg` all Slot, and (ii) the initial software handlers such as `Sw_Task_Listener` and `Signal_Recev`.

6. Client support library API

The client support library provides a lightweight API that can be used by programmers for developing FPGA-accelerated applications with Fred-Linux. Both a C and a Python version of the library are available. For simplicity, only the former is discussed in the following. Note that, since SW-tasks are completely decoupled from fred-server through UNIX domain sockets, additional client APIs for other languages can be easily developed as long as they follow the same communication protocol with fred-server.

Listing 4 reports the functions composing the client support library API. The `fred_init` function initiates the communication with fred-server by initializing an opaque handler of type `struct fred_data` that holds the state of the connection. After the initialization phase, a SW-task can request the association with one or more HW-tasks using the `fred_bind` function. Such a function takes as input the `id` of the HW-task and initializes an opaque handler `fred_hw_task`, which contains a set of references to the data buffers used to share the data between the SW-task (i.e., the current process or thread) and the HW-task. These buffers can be mapped into the address space of the calling SW-task using the `fred_map_buff` function, which takes as input the `fred_hw_task` handle of the HW-task and the index of the buffer, returning a pointer to the mapped buffer. The service functions `fred_get_buffs_count` and `fred_get_buff_size` can be used to query the number and the size of the buffers used by an HW-task, respectively.

Once the SW-task has completed its initialization phase, binding with its associated HW-tasks and mapping the respective data buffers, it can proceed with its computations. Each computation starts with the SW-task filling the shared buffers with the input data and proceeds by sequentially calling one or more HW-tasks using the `fred_accel` function. The `fred_accel` is a blocking function that suspends

Listing 4: Client support library API functions.

```

1 struct fred_data;
2 struct fred_hw_task;
3
4 /*-----*/
5
6 int fred_init(struct fred_data **self);
7
8 int fred_bind(struct fred_data *self, struct fred_hw_task
9             **hw_task, uint32_t hw_task_id);
10 int fred_accel(struct fred_data *self, const struct
11              fred_hw_task *hw_task);
12 void fred_free(struct fred_data *self);
13
14 /*-----*/
15
16 int fred_async_accel(struct fred_data *self, const struct
17                   fred_hw_task *hw_task);
18 int fred_async_wait_for_accel(struct fred_data *self);
19
20 /*-----*/
21
22 int fred_get_buffs_count(const struct fred_data *self,
23                        struct fred_hw_task *hw_task);
24 ssize_t fred_get_buff_size(const struct fred_data *self,
25                          struct fred_hw_task *hw_task, int buff_idx);
26
27 /*-----*/
28 void *fred_map_buff(const struct fred_data *self, struct
29                   fred_hw_task *hw_task, int buff_idx);
30 void fred_unmap_buff(const struct fred_data *self, struct
31                    fred_hw_task *hw_task, int buff_idx);

```

the SW-tasks until the invoked HW-task completes its execution. Alternatively, the SW-task can asynchronously call one or more HW-tasks using the `fred_async_accel` function. In this way, the SW-task can proceed with its execution in parallel with the HW-task. Then, it can synchronize with the HW-task using the `fred_async_wait_for_accel` function, which suspends the calling SW-task until the HW-task completes and the result becomes available. After the HW-task completion, the SW-task will resume its execution and can retrieve the data processed by the HW-task by accessing the shared buffers as regular memory. It is worth noting that the design of the HW-task does not change depending on the selected acceleration mode (synchronous or asynchronous). Finally, during the system shutdown phase, the SW-task can unmap all the shared buffers using the `fred_unmap_buff` and close the session with fred-server by calling the `fred_free` function. Listing 5 shows the pseudo-code of a SW-task implemented using the C API provided by the client support library. For the sake of clarity, the SW-task only uses a single HW-task and the code to handle errors is omitted.

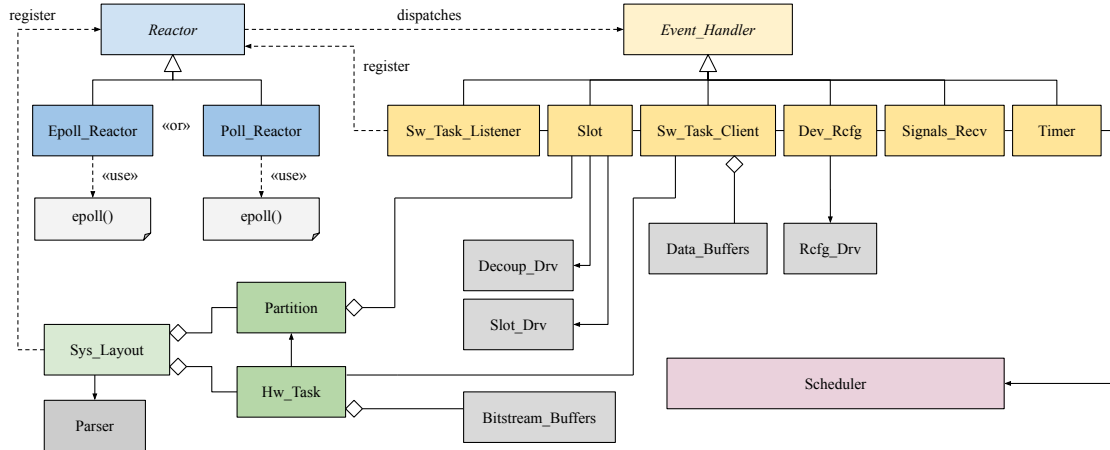


Figure 6: Internal components of the fred-server.

7. Case study application

Listing 5: Pseudo-code stub of a SW-task using the C API.

```

1 void sw_task(void *args)
2 {
3     struct timespec ts;
4     int period_ms = <task_period>;
5
6     struct fred_data *fred;
7     struct fred_hw_task *hw_task;
8     uint32_t hw_task_id = <hw_task_id>;
9
10    void *buff_in = NULL;
11    void *buff_out = NULL;
12
13    /* Initialize communication and bind a HW-task */
14    fred_init(&fred_data);
15    fred_bind(fred_data, &hw_task, hw_task_id);
16
17    /* Map the buffers */
18    buff_in = fred_map_buff(fred, hw_task, 0);
19    buff_out = fred_map_buff(fred, hw_task, 1);
20
21    /* Get current time */
22    clock_gettime(CLOCK_MONOTONIC, &ts);
23    /* Set next activation */
24    time_add_ms(&ts, period_ms);
25
26    while (true) {
27        /* Fill input buffer */
28        buff_in[i] = <...>
29
30        /* Call the HW-task */
31        fred_accel(fred_data, hw_task);
32
33        /* Read output data buffer */
34        <...> = buff_out[i]
35
36        /* Sleep until next activation */
37        clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &t,
38                        NULL);
39        /* Set next activation */
40        time_add_ms(&t, period_ms);
41    }
42 }

```

This section presents a case study application that makes use of Fred-Linux for speeding up real-time processing of live images acquired by a USB webcam and integer matrix multiplications. The case study application has been implemented in C++11 using Qt and V4L on top of Xilinx Petalinux, and has been executed and tested on the Digilent’s Zybo board, which includes a Zynq-7010 SoC supported by 512 MB of DDR3 memory.

The application comprises four *processing functions* implemented as both HW-tasks and functionally-equivalent software procedures for evaluation purposes. Three processing functions are image processing filters (FastX, gradient map, and Sobel) designed using the popular OpenCV library. The HW-tasks filters are built with Vivado HLS using the available OpenCV subset, while the equivalent software implementations are built using the typical OpenCV C++ API. The filters are implemented as a stack of OpenCV functions inspired by Xilinx’s guidelines provided in [45]. Figure 7 shows the output of the image filters using a test image as input. Finally, the last processing function is an integer matrix multiplier implemented in both HLS and C++ using the naive $O(n^3)$ algorithm.

The amount of logic resources required to allocate all HW-tasks in a static fashion (i.e., deployed altogether on the FPGA at the same time) exceeds the amount of resources available on the physical FPGA. Hence, using hardware acceleration for all tasks would be unfeasible with standard approaches. Fred-Linux allows to virtually extend the amount of logic of resources hence making possible to accommodate all the accelerators in timesharing.

7.1. Case study architecture

From a system perspective, the application is composed of four SW-tasks and four HW-tasks. Each SW-task is a cyclic thread that calls a processing function during each job. Each HW-task implements a hardware processing function. The SW-tasks can operate in two modes: (i) software

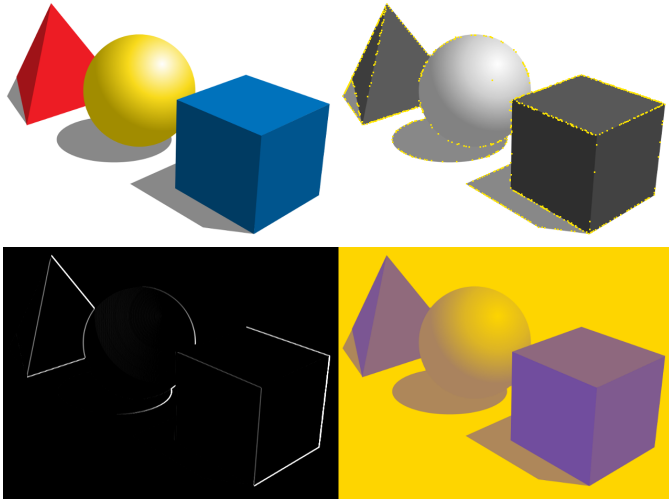


Figure 7: Test image processed using filter processing functions. Clockwise from the top: Test image, FastX filter, gradient map (Gmap) filter, Sobel filter.

mode and (ii) hardware mode. In the software mode, the SW-task processes the data using the software implementation of the processing function; in hardware mode, the SW-task relies on hardware acceleration by invoking the corresponding HW-task to perform the computation. SW-tasks are scheduled using the `SCHED_FIFO` policy, which implements static-priority scheduling on Linux. Hence, they cannot be preempted by regular processes which are scheduled using the default `SCHED_OTHER` class.

In the default configuration, the image filters are configured for processing images of 640×480 pixels with 24-bit color depth, while the matrix multiplier performs 30 multiplications of 64×64 matrices. The hardware implementations of the processing functions have been wrapped within the Fred-Linux standard HW-task interface presented in Listing 2 and translated into RTL implementations using Vivado HLS 2017.4. The equivalent software versions are compiled as regular C++ functors or functions using GCC 6.2.1.

The input data for the image filters are acquired through a USB webcam using the V4L framework. A schematic representation of the internal architecture of the application is presented in Figure 8. The frame grabber thread copies the frames acquired by the webcam into a shared buffer implemented as a *cyclic asynchronous buffer* (CAB) [46]. The CAB mechanism is designed to support asynchronous lock-free communication between cyclic activities with different periods. In this way, the image processing SW-tasks can read the frames from the buffer without blocking, even when having different periods. After reading the image from the CAB, each SW-task processes the input frame depending on the current processing mode. If the SW-task is set in software processing mode, the frames are processed using the OpenCV software procedure, and the output is directed to a Qt image (QImage) buffer. If the SW-task is set for hardware processing, the frames are copied from the

cyclic buffer to the input buffer of the correspondent image processing HW-task. Once the copy is completed, the HW-tasks execution request is sent to fred-server, and the SW-task is suspended. After the completion of a HW-task, the calling SW-task resumes its execution and retrieves the processed frame from the HW-task's output buffer, which is associated with a QImage object to avoid another additional copy. Finally, independently from the operating mode, the resulting image is stored in a QImage buffer that can be passed to the Qt window component to be displayed.

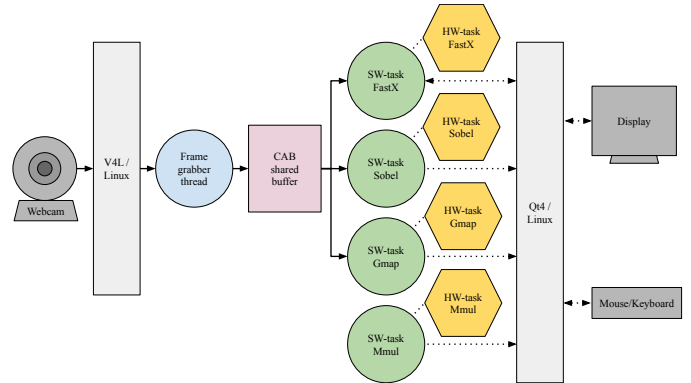


Figure 8: Structure of the case study application.

7.1.1. Programmable logic partitioning

The Zynq PL FPGA fabric is divided into a static region and a reconfigurable region according to the Fred-Linux support design described in Section 4.1. The static region contains a set of AXI Interconnects and other support modules like a video output module. In contrast, the reconfigurable region is organized in partitions and slots for dynamically hosting the HW-tasks. More specifically, the reconfigurable region is divided into two partitions, each containing one slot. The first partition P_0 contains roughly 32% of the total logic resources (5600 LUTs), while the second partition P_1 includes 14% of the total logic resources (2400 LUTs). The remaining resources, 9600 LUTs, corresponding to approximately 54% of the total, are reserved for the static region. A graphical representation of the partitioning is presented in Figure 9. The design has been manually floorplanned using Vivado 2017.4. The total resource distribution is slightly more uneven since special-purpose cells like DSPs and BRAMs are not homogeneously distributed on the fabric.

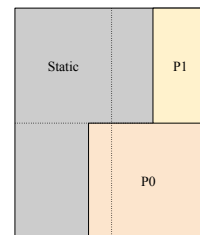


Figure 9: Representation of the FPGA fabric partitioning.

Exec. time	FastX	Gmap	Sobel	Mmul
SW avg (ms)	58.222	55.521	68.823	65.080
SW max (ms)	62.864	56.690	71.119	70.112
HW avg (ms)	4.905	4.770	4.864	23.662
HW max (ms)	5.068	4.879	4.976	23.748
Avg. speed up	11.869	11.639	14.146	2.751

Table 1: Results of speedup evaluation experiments.

The FastX and the matrix multiplier HW-tasks are associated to the largest partition P_0 as they require more resources. The Sobel and Gmap HW-tasks are associated to partition P_1 . Since both partitions are composed of a single slot, only one bitstream is required for each HW-task.

8. Performance evaluation

This section presents the results of a set of experiments aimed at evaluating the performance of Fred-Linux using the case study application on a Xilinx Zynq-7000 platform. Furthermore, this section also reports on an experimental evaluation to investigate on the reconfiguration throughput on the modern Xilinx Ultrascale+ platform using the standard Linux support provided by the vendor.

8.1. Speedup evaluation experiments

This first set of experiments has been carried out to evaluate the speedup achieved for each processing function by means of hardware acceleration with respect to pure software implementations. One experiment per processing function has been performed by configuring the case-study application to run only a single SW-task to avoid interferences. Each experiment consists in running the SW-task both in software and hardware mode for 10×10^3 times.

Under both operating modes, the SW-task is profiled using a logic analyzer connected to the GPIO pins controlled by the ARM core inside the PS. It is worth noting that, even if there is no contention among SW-tasks since only one is running, the system load caused by the Qt framework, fred-server, and Linux is still present. The results of the experiments are summarized in Table 1. It is worth observing that execution times of the hardware implementations of the image processing functions are similar despite implementing different algorithms. These similarities can be explained by observing that the execution time of image filter HW-tasks is dominated by the time required to access memory and that they process images of the same size, i.e., 640×480 pixels with 24-bit color depth, hence, they move the same amount of data from and to the DRAM system memory.

8.2. System acceleration experiment

A second experiment has been then carried out to evaluate the performance improvement achievable with DPR-enabled hardware acceleration by means of Fred-Linux considering the whole application. For this purpose, the application is configured to concurrently run with all four

SW-tasks at the same time. In order to evaluate the system speedup enabled by Fred-Linux, the experiment is composed of two separate runs. In the first run, all the SW-tasks are configured to run in software mode while, in the second run, all SW-tasks are set for running in hardware mode.

The main difference with respect to the first set of experiments is that now all SW-tasks execute and perform acceleration request to fred-server concurrently. Hence, when a SW-task running in hardware mode (i.e., calling an HW-task) performs an acceleration request, it can experience a delay because there are no available free slots or the reconfiguration interface is busy. The parameters of all software activities involved in this test, including SW-tasks, are summarized in Table 2. The results of the experiment are reported in Table 3 comparing the observed response times of the SW-tasks while running in software and hardware modes in a 30-minute run. Figure 10 presents a distribution of the response times, while Figure 11 shows a distribution of the reconfiguration times observed during the test.

Overall, the results of this practical evaluation show that the resource virtualization mechanism provided by Fred-Linux allows improving the performance of case study application through hardware acceleration using a set of HW-tasks that are impossible to statically allocate on the FPGA due to the unavailability of a sufficient amount of resources.

Activity name	Relative Priority	Period [ms]
fred-server (process)	0	-
Qt event loop thread	0	-
Frame grabber thread	1	33.3
Plain image copy thread	2	50
SW-task Sobel (thread)	3	80
SW-task Gmap (thread)	3	80
SW-task FastX (thread)	3	120
SW-task Mmul (thread)	3	120

Table 2: Priorities and periods of software activities.

Activity	Response time [ms]			
	SW avg	HW avg	SW max	HW max
SW-task Sobel	180.024	48.978	616.779	111.154
SW-task Gmap	161.861	50.173	731.724	96.756
SW-task FastX	118.670	65.125	515.957	106.558
SW-task Mmul	102.304	53.017	274.588	108.384

Table 3: Comparison of the observed response times for SW-tasks in both software and hardware modes.

8.3. Common interface resource consumption

The FRED common interface requires 627 LUTs, 894 Flip-Flops, 1 BRAM tile, and 0 DSPs. This accounts for 3.65 % of LUTs, 2.6 % of Flip-Flops, 0 % of DSPs, and 3.33 % of BRAM tiles available on the Zynq-7010 SoC used in this evaluation. Please note that the interface is not

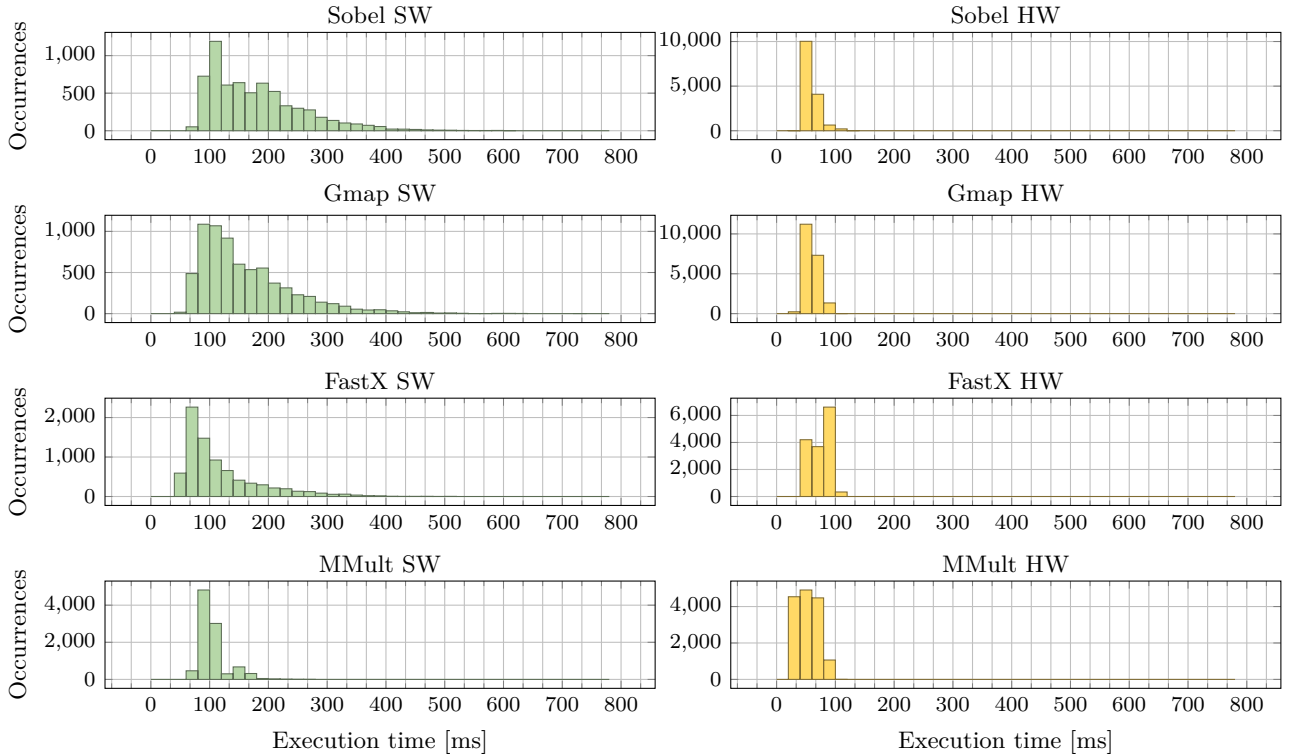


Figure 10: Comparison of response times distributions for SW-tasks in software and hardware mode.

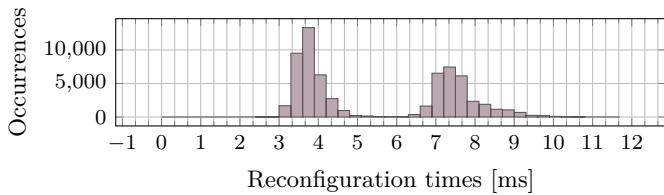


Figure 11: Distribution of the reconfiguration times.

dependent on the memory used for sharing data. From the HW-tasks perspective, on-chip RAM and off-chip DRAM are both accessible in the global address space using an AXI master interface connected to the AXI slave ports exported by the processing system.

8.4. Overhead evaluation

The overhead introduced by the implementation of Fred-Linux presented in this work has also been evaluated. In particular, the overhead introduced by fred-server when serving requests issued by SW-tasks has been measured while running the case-study application. Figure 12 presents the distribution of the execution times measured for each activation of the fred-server process during a 30-minute run. The longest-observed execution times is 243.7 μ s, while the average execution time resulted in 72.4 μ s. These measurements account for both the inter-process communication and the server process execution. It is worth noting that the overhead introduced by fred-server

does not depend on the amount of data shared among SW-tasks and HW-tasks.

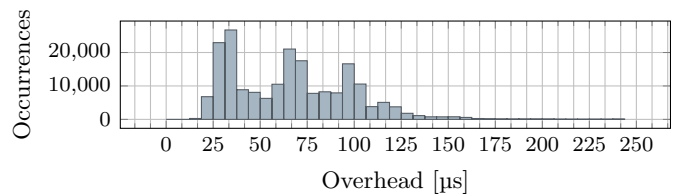


Figure 12: Distribution of the overhead introduced by the fred-server.

8.5. Reconfiguration throughput evaluation

To better assess the viability of the approach used by FRED, which is based on an intense usage of DPR, a further experiment has been conducted to evaluate the reconfiguration throughput on the modern Zynq Ultrascale+ platform by Xilinx. The PCAP reconfiguration interface has been used. For each reconfiguration, the FPGA manager driver running on the ARM Cortex-53 cores of the Zynq Ultrascale+ issues a reconfiguration request to ATF using the EEMI interface. In turn, ATF forwards the request to the PMU using an inter-processor interrupt. The PMU MicroBlaze core serves the request by programming the CSU's DMA for transferring the required bitstream. It is worth noting that the Linux kernel cannot directly access the CSU registers under normal conditions. This is because the Linux kernel runs at exception level 1 (EL1)

non-secure state, while the CSU registers can normally be accessed only at exception level 3 (EL3) secure state. The need for such a complex architecture arises from the heterogeneous nature of the platform. The Zynq UltraScale+ SoC includes multiple processing cores that can independently execute software components with different safety and trustworthiness levels. If any software could autonomously perform actions that can affect the entire platform (e.g., reconfiguring the FPGA fabric), it could jeopardize the safety of the whole system. For this reason, on the Zynq UltraScale+ the responsibility of platform management has been delegated to the PMU, which acts as a trusted central component. This experiment aims at characterizing the reconfiguration throughput while traversing this complex path that involves the PMU.

The evaluation has been performed on the ZCU102 board, which includes the Zynq UltraScale+ XCZU9EG chip. In order to assess how the reconfiguration throughput varies with respect to the size of the slot, the same HW-task has been cloned and implemented in three different partitions containing a single slot. These partitions represent respectively 1/16, 1/8, and 1/4 of the PL fabric resources available on the XCZU9EG chip. The results of this evaluation are reported in Figure 13. The bars above and below the samples represent the minimum and maximum observed throughput values over more than 500×10^3 reconfiguration events. A minimum throughput of 635 MiB/s has been observed while reconfiguring the smallest slot (1/16), while the maximum throughput of 673 MiB/s has been observed during the reconfiguration of the larger slot (1/4). These results can be explained by considering the fixed overhead introduced by both the IPI-based communication mechanism and the *xilfpga* library for configuring the CSU DMA. In general, the reconfiguration time can be modeled as the sum of a fixed overhead component and a variable component that linearly grows with the bitstream size. Hence, the fixed overhead component becomes proportionally less relevant for larger bitstreams. This evaluation confirms that the PCAP reconfiguration throughput on the Zynq UltraScale+ has been vastly improved over the previous Zynq-7000 platforms, where it was limited to 145 MiB/s [20]. This improvement makes partial reconfiguration even more profitable for real-time embedded systems, as leveraged by Fred-Linux.

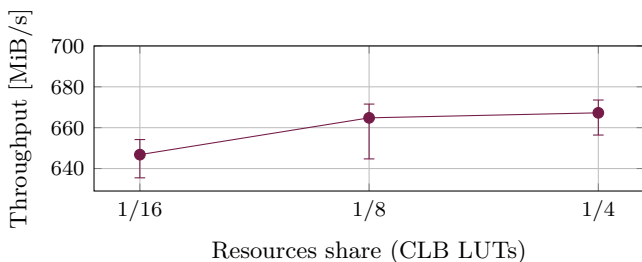


Figure 13: Reconfiguration throughput with respect to the number of PL resources available on the Zynq UltraScale+ XCZU9EG.

9. Conclusions

This paper presented a novel implementation of the FRED framework for GNU/Linux. The proposed implementation has been discussed in detail, highlighting novel features such as a completely redesigned software architecture as well as the support for asynchronous acceleration, watchdog timers, and Zynq UltraScale+ MPSoCs through the FPGA manager. The implementation has been tested with a case-study application using standard components such as OpenCV and Video for Linux. Experimental results show that real-world applications can benefit from time-predictable FPGA-based hardware acceleration in a GNU/Linux feature-rich operating system environment. Future work should investigate the possibility of integrating the proposed implementation on top of a hypervisor with strong isolation capabilities, providing a safe platform for developing Cyber-Physical Systems [47]. Further developments include improving integration with upcoming releases of Xilinx’s FPGA manager driver and minimizing kernel-level code by moving memory buffers allocation and management in userspace.

References

- [1] A. Biondi, A. Balsini, M. Pagani, E. Rossi, M. Marinoni, G. Buttazzo, A Framework for Supporting Real-Time Applications on Dynamic Reconfigurable FPGAs, in: Proc. of the IEEE Real-Time Systems Symposium (RTSS 2016), 2016, pp. 1–12.
- [2] M. Pagani, A. Balsini, A. Biondi, M. Marinoni, G. Buttazzo, A linux-based support for developing real-time applications on heterogeneous platforms with dynamic fpga reconfiguration, in: 2017 30th IEEE International System-on-Chip Conference (SOCC), IEEE, 2017, pp. 96–101.
- [3] B. B. Seyoum, M. Pagani, A. Biondi, S. Balleri, G. Buttazzo, Spatio-Temporal Optimization of Deep Neural Networks for Reconfigurable FPGA SoCs, IEEE Transactions on Computers (01) (5555) 1–1. doi:10.1109/TC.2020.3033730.
- [4] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, K. Vissers, Finn: A framework for fast, scalable binarized neural network inference, in: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2017, pp. 65–74.
- [5] Xilinx, Pynq - python productivity for zynq. URL <http://www.pynq.io/>
- [6] Fred-linux repository. URL <https://gitlab.retis.sssup.it/m.pagani/fred-docs>
- [7] F. Dittmann, S. Frank, Hard real-time reconfiguration port scheduling, in: Proc. of the Conference on Design, Automation and Test in Europe (DATE), 2007.
- [8] F. Duhem, F. Muller, P. Lorenzini, FaRM: Fast Reconfiguration Manager for Reducing Reconfiguration Time Overhead on FPGA, in: Reconfigurable Computing: Architectures, Tools and Applications, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 253–260.
- [9] M. Damschen, L. Bauer, J. Henkel, CoRQ: Enabling Runtime Reconfiguration Under WCET Guarantees for Real-Time Systems, IEEE Embedded Systems Letters 9 (3) (2017) 77–80. doi:10.1109/LES.2017.2714844.
- [10] G. Valente, T. Di Mascio, G. D’Andrea, L. Pomante, Dynamic partial reconfiguration profitability for real-time systems, IEEE Embedded Systems Letters (2020) 1–1doi:10.1109/LES.2020.3004302.
- [11] L. Pezzarossa, A. T. Kristensen, M. Schoeberl, J. Sparsø, Using dynamic partial reconfiguration of FPGAs in real-time systems,

- Microprocessors and Microsystems 61 (2018) 198–206. doi:
<https://doi.org/10.1016/j.micpro.2018.05.017>.
- [12] J. Goossens, X. Poczekajko, A. Paolillo, P. Rodriguez, ACEPTOR: A Model and a Protocol for Real-Time Multi-Mode Applications on Reconfigurable Heterogeneous Platforms, in: Proceedings of the 27th International Conference on Real-Time Networks and Systems, RTNS '19, Association for Computing Machinery, New York, NY, USA, 2019, p. 209–219. doi:10.1145/3356401.3356420.
 URL <https://doi.org/10.1145/3356401.3356420>
- [13] K. Danne, M. Platzner, Periodic real-time scheduling for FPGA computers, in: Proc. of the 3rd Int. Workshop on Intelligent Solutions in Embedded System, 2005.
- [14] S. Saha, A. Sarkar, A. Chakrabarti, Scheduling dynamic hard real-time task sets on fully and partially reconfigurable platforms, IEEE Embedded Systems Letters 7 (1) (2015) 23–26.
- [15] S. Saha, A. Sarkar, A. Chakrabarti, R. Ghosh, Co-Scheduling Persistent Periodic and Dynamic Aperiodic Real-Time Tasks on Reconfigurable Platforms, IEEE Transactions on Multi-Scale Computing Systems 4 (1) (2018) 41–54. doi:10.1109/TMSCS.2017.2691701.
- [16] A. Rodríguez, J. Valverde, J. Portilla, A. Otero, T. Riesgo, E. De la Torre, FPGA-Based High-Performance Embedded Systems for Adaptive Edge Computing in Cyber-Physical Systems: The ARTICo3 Framework, Sensors 18 (6). doi:10.3390/s18061877.
 URL <https://www.mdpi.com/1424-8220/18/6/1877>
- [17] E. Lübbers, M. Platzner, ReconOS: Multithreaded Programming for Reconfigurable Computers, ACM Transactions on Embedded Computing Systems 9 (1) (2009) 8:1–8:33.
- [18] E. Lübbers, M. Platzner, Cooperative multithreading in dynamically reconfigurable systems., in: Proc. of the Int. Conference on Field Programmable Logic and Applications (FPL), 2009.
- [19] X. Iturbe, K. Benkrid, C. Hong, A. Ebrahim, R. Torrego, T. Arslan, Microkernel Architecture and Hardware Abstraction Layer of a Reliable Reconfigurable Real-Time Operating System (R3TOS), ACM Transactions on Reconfigurable Technology and Systems 8 (1) (2015) 5:1–5:35.
- [20] M. Pagani, M. Marinoni, A. Biondi, A. Balsini, 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.
- [21] R. T. E. Ltd., Freertos real-time operating system.
 URL <http://www.freertos.org/>
- [22] H. K.-H. So, R. Brodersen, A Unified Hardware/Software Runtime Environment for FPGA-based Reconfigurable Computers Using BORPH, ACM Transactions on Embedded Computing Systems 7 (2) (2008) 14:1–14:28.
- [23] The linux driver implementer's api guide, fpga subsystem.
 URL <https://www.kernel.org/doc/html/latest/driver-api/fpga/fpga-mgr.html>
- [24] B. B. Seyoum, A. Biondi, G. C. Buttazzo, Flora: Floorplan optimizer for reconfigurable areas in fpgas, ACM Trans. Embed. Comput. Syst. 18 (5s). doi:10.1145/3358202.
 URL <https://doi.org/10.1145/3358202>
- [25] M. Hassan, R. Pellizzoni, Bounding DRAM interference in COTS heterogeneous MPSoCs for mixed criticality systems, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 37 (11) (2018) 2323–2336.
- [26] D. Casini, A. Biondi, G. Nelissen, G. Buttazzo, A holistic memory contention analysis for parallel real-time tasks under partitioned scheduling, in: 2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), IEEE, 2020, pp. 239–252.
- [27] F. Restuccia, M. Pagani, A. Biondi, M. Marinoni, G. Buttazzo, Modeling and analysis of bus contention for hardware accelerators in fpga socs, in: 32nd Euromicro Conference on Real-Time Systems (ECRTS 2020), Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
- [28] Xilinx, Vivado Design Suite User Guide: Dynamic Function eXchange.
- [29] C. Beckhoff, D. Koch, J. Torresen, Go Ahead: A Partial Reconfiguration Framework, in: Proc. of the 20th Annual IEEE Int. Symposium on Field-Programmable Custom Computing Machines, 2012.
- [30] D. Koch, Partial Reconfiguration on FPGAs: Architectures, Tools and Applications, Springer-Verlag New York, 2012.
- [31] Xilinx, Zynq-7000 AP SoC Technical Reference Manual.
- [32] Xilinx, Zynq UltraScale+ Device - Reference Manual, uG1085.
- [33] F. Restuccia, M. Pagani, A. Biondi, M. Marinoni, G. Buttazzo, Modeling and Analysis of Bus Contention for Hardware Accelerators in FPGA SoCs (Artifact), Dagstuhl Artifacts Series 6 (1) (2020) 4:1–4:3. doi:10.4230/DARTS.6.1.4.
 URL <https://drops.dagstuhl.de/opus/volltexte/2020/12394>
- [34] M. Pagani, E. Rossi, A. Biondi, M. Marinoni, G. Lipari, G. Buttazzo, A Bandwidth Reservation Mechanism for AXI-Based Hardware Accelerators on FPGAs, in: S. Quinton (Ed.), 31st Euromicro Conference on Real-Time Systems (ECRTS 2019), Vol. 133 of Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2019, pp. 24:1–24:24. doi:10.4230/LIPIcs.ECRTS.2019.24.
 URL <http://drops.dagstuhl.de/opus/volltexte/2019/10761>
- [35] F. Restuccia, A. Biondi, M. Marinoni, G. Cicero, G. Buttazzo, Axi hyperconnect: a predictable, hypervisor-level interconnect for hardware accelerators in fpga soc, in: 2020 57th ACM/IEEE Design Automation Conference (DAC), IEEE, 2020, pp. 1–6.
- [36] Xilinx Inc., AXI Interconnect, LogiCORE IP Product Guide, pG059.
- [37] Xilinx Inc., SmartConnect, LogiCORE IP Product Guide, pG247.
- [38] B. Seyoum, M. Pagani, A. Biondi, G. Buttazzo, Automating the design flow under dynamic partial reconfiguration for hardware-software co-design in fpga soc, in: Proceedings of the 36th Annual ACM Symposium on Applied Computing, SAC '21, Association for Computing Machinery, New York, NY, USA, 2021, p. 481–490. doi:10.1145/3412841.3441928.
 URL <https://doi.org/10.1145/3412841.3441928>
- [39] Xilinx, Xilinx Standalone Library Documentation, XilFPGA Library.
- [40] D. C. Schmidt, Reactor: An object behavioral pattern for concurrent event demultiplexing and dispatching, 1995.
- [41] A. Petersen, Patterns in C, 2005.
- [42] R. C. Martin, Design principles and design patterns, Object Mentor 1 (34) (2000) 597.
- [43] M. Samek, Object-Oriented Programming in C, 2019.
- [44] A.-T. Schreiner, Object oriented programming with ansi-c.
- [45] Xilinx, Accelerating OpenCV Applications with Zynq-7000 All Programmable SoC using Vivado HLS Video Libraries.
- [46] G. C. Buttazzo, Hartik: A real-time kernel for robotics applications, in: Real-Time Systems Symposium, 1993., Proceedings., IEEE, 1993, pp. 201–205.
- [47] A. Biondi, F. Nesti, G. Cicero, D. Casini, G. Buttazzo, A safe, secure, and predictable software architecture for deep learning in safety-critical systems, IEEE Embedded Systems Letters 12 (3) (2019) 78–82.