Proc. of the 30th IEEE International System-on-Chip Conference (SOCC 2017), Munich, Germany, September 5-8, 2017.

A Linux-based Support For Developing Real-Time Applications On Heterogeneous Platforms with Dynamic FPGA Reconfiguration[‡]

Marco Pagani*†, Alessio Balsini*, Alessandro Biondi*, Mauro Marinoni*, Giorgio Buttazzo*
Scuola Superiore Sant'Anna, Pisa, Italy* Université Lille 1, Lille, France†
Email: {marco.pagani, alessio.balsini, alessandro.biondi, mauro.marinoni, giorgio.buttazzo}@santannapisa.it

***Abstract*—Heterogeneous computing platforms including both processors and field programmable gate arrays (FPGAs) represent an attractive solution for balancing software flexibility with high performance and energy efficiency of custom hardware modules. Furthermore, the dynamic partial reconfiguration (DPR) capabilities of modern FPGAs allow virtualizing the available area to support several hardware modules in time sharing, hence making them even more attractive. Such a feature is exploited by the FRED framework, recently proposed to support the development of real-time applications upon such platforms.**

This paper presents an implementation of the FRED framework for the Linux operating system over the Zynq-7000 platform produced by Xilinx. Design solutions for managing hardware accelerators are first discussed. Then, a software architecture for Linux is presented, which comprises (i) support for shared-memory communication with hardware accelerators, (ii) an improved driver to handle the FPGA reconfiguration and (iii) a scheduler for requests of hardware acceleration. The proposed solution allows exploiting the enormous number of software systems available for Linux (such as drivers, libraries, communication stacks, etc.) and the typical programming flexibility of software, while relying on predictable hardware acceleration of heavy computations.

I. Introduction

Embedded computing platforms are evolving towards heterogeneous architectures that integrate multiple processing elements of different nature, as classical central processing units (CPUs), general-purpose computing on graphics processing units (GPGPUs), and field programmable gate arrays (FPGAs). Such platforms allow balancing the flexibility of software systems with the advantages of a highly parallel custom hardware acceleration, thus achieving a consistent speed-up with a contained energy consumption. In addition, modern FPGAs offer a dynamic partial reconfiguration (DPR) capability, which allows the user to re-program a part of the FPGA fabric while the remaining logic resources continue to operate without interruption. By looking at the FPGAs produced in the last 15 years, there is a clear evolution trend where reconfiguration times have been progressively decreased, approaching a reconfiguration throughput of 800 MB/s for today's platforms (e.g., Xilinx Ultrascale+). Such a faster DPR capability enables the possibility of virtualizing the FPGA area to support several hardware modules in time-sharing, hence making these devices even more attractive.

Recently, Biondi et al. [1] proposed a programming framework, named FRED, to support the development of real-time applications upon such heterogeneous platforms, providing a predictable infrastructure that can ensure bounded delays when requesting a dynamically-reconfigured hardware accelerator. However, only a simple proof-of-concept on top FreeRTOS is presented.

The present paper proposes the implementation of the FRED framework on the Linux operating system addressing several challenges, such as the architectural support for the accelerators, the reconfiguration and communication mechanisms, the implementation of the FRED scheduler, and the synchronization mechanisms between software and hardware tasks. In particular, it presents a software architecture for Linux composed of (i) a kernel module for implementing shared-memory communication with hardware accelerators, (ii) a driver to handle the FPGA reconfiguration, and (iii) a user-space server process to schedule the requests for hardware acceleration.

The rest of the paper is organized as follows. Section II reviews the related works. Section III outlines the main concepts behind FRED. Section IV illustrates the proposed FPGA support (hardware design). Section V describes the Linux implementation of FRED. Section VI reports some experimental results carried out on the Zynq platform. Section VII concludes the paper and presents some future work.

II. Related Work

A timing analysis of real-time applications on FPGA-based platforms is still an open issue, since most of the proposed approaches are based on simplistic assumptions that do not match real platforms. The model proposed by Danne and Platzner [2] requires HW-task preemptability and is oversimplistic, considering negligible reconfiguration time and no allocation constraints. The model proposed by Saha et al. [3] also considers preemtable HW-tasks on homogeneous partitions reallocating the FPGA at every HW-task termination. This induces a waste on the FPGA area and increases the pessimism in the analysis. Dittmann and Frank [4] addressed the problem of scheduling reconfiguration requests using a single shared non-preemptive reconfiguration interface, as the Xilinx ICAP port. Biondi et al. [1] proposed the FRED framework, which is a predictable infrastructure to support DPR of hardware accelerators. Being based on a more detailed platform model, it can be implemented on actual platforms.

To exploit DPR features of FPGA platforms from an operating system perspective, few software solutions have been proposed. Lübbers and Platzner [5] 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 [6]. Interactions among threads are managed by common POSIX-like abstractions (e.g., semaphores, shared memory). R3TOS is an operating system proposed by Iturbe

[‡]The authors would like to thank Daniel Bristot de Oliveira, Luca Abeni and Tommaso Cucinotta for their support in the development of this work.

et al. [7] supporting dynamic allocation of HW-tasks on an FPGA without the need of 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. [8] proposed a prototype implementation of the FRED framework over the FreeRTOS [9] kernel to show the applicability of the approach.

Also the Linux community has shown interest in the exploitation of the FPGAs features. However, the current mainline kernel only provides a simple support for the reconfiguration interface. So and Brodersen proposed BORPH [10], 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.

The present work overcomes some limitations of the current state of the art providing: (i) the implementation of a framework designed to increase predictability of application exploiting FPGA acceleration; (ii) an implementation that does not limit HW-tasks to specific paradigms (e.g., stream processing, data flow); (iii) an efficient use of HW resources (i.e., improved reconfiguration interface driver, zero-copy data transfer mechanisms), (iv) a scalable implementation with respect to the number of HW-tasks in the system; and (v) a seamless integration in the Linux kernel.

III. Essential Background: The FRED Framework

This section briefly reviews the FRED framework proposed in [1], on which this work is based. The FRED framework considers a heterogeneous computing system consisting of (at least) *one* processor and a DPR-enabled FPGA fabric, both sharing a common memory. Two types of computational activities are managed:

- *software tasks* (SW-tasks): they are computational activities running on the processors; and
- *hardware tasks* (HW-tasks): they are hardware accelerators that can be configured and executed on the FPGA.

SW-tasks can speedup parts of their computation by requesting the execution of HW-tasks. To ensure predictability when reconfiguring the FPGA while minimizing the overhead related to HW-task allocation, FRED relies on a *static partitioning* of the FPGA area. That is, the area is split into n_P *partitions*, each composed of n_k^S equal *slots* ($k = 1, ..., n_P$). Logic blocks are not shared among partitions nor among slots.

Hardware Tasks. Each HW-task has a static affinity to a single partition and can execute only if it has been programmed into one of its slots. HW-tasks execute in a non-preemptive manner. Each slot can be reconfigured at run-time by means of an *FPGA reconfiguration interface* (FRI) and can accommodate at most one HW-task. As typical for most real-world platforms (e.g., [11], [12]), the FRI (i) can reconfigure a slot without affecting the execution of the HW-tasks currently programmed in other slots; (ii) is a peripheral device external to the processor (e.g., like a DMA [13]) and hence does not consume processor cycles to reconfigure slots; and (iii) can program at most one slot at a time.

Software Tasks. Each SW-task uses a set of HW-tasks by alternating execution phases with *suspension* phases where the SW-task is descheduled to wait for the completion of the requested HW-task. The same HW-task cannot be used by more than one SW-task. Each SW-task is cyclically released, thus generating an infinite sequence of execution instances (jobs). SW-tasks are also subject to timing constraints, meaning that each of its jobs must complete within a *deadline* relative to its activation. A sample schedule of a SW-task that uses two HW-tasks is illustrated in Figure 1.

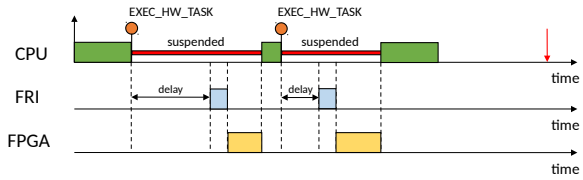

Figure 1. Sample schedule of a SW-task using two HW-tasks. Up-arrows and down-arrows denote the release and deadline of the SW-task, respectively.

A. Scheduling Infrastructure

The FRED framework comes with a scheduling mechanism to handle the contention of the FRI and the FPGA slots. Such a mechanism is based on a multi-level queueing structure as illustrated in Figure 2, which includes (i) n_p *partition queues* (one for each partition), needed to schedule the requests for HW-tasks with affinity to the same partition, and (ii) a *FRI queue* to schedule the reconfiguration 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 predictable delays incurred by HW-task requests, which have also been bounded via a response-time analysis [1]. Please refer to [1] for a precise formalization of the scheduling rules and additional details.

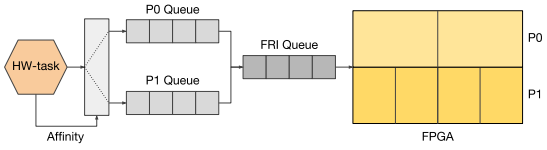

Figure 2. Multi-level queuing structure for scheduling HW-tasks. The FPGA area is divided into two partitions P_0 and P_1.

B. Communication between SW and HW tasks

The FRED framework employs a *shared-memory* communication paradigm between software and hardware tasks. In contrast to other approaches that store the input/output data into private memory areas within the FPGA slots, the solution adopted in FRED allows decoupling the time a HW-task must hold a slot from the scheduling delays of SW-tasks. This property facilitates the derivation of bounds on the delay incurred by

the SW-tasks when requesting HW-tasks. Please refer to [1] for further details.

IV. FPGA SUPPORT

This section proposes a design to support the FRED framework on the Xilinx Zynq-7000 System-on-Chip (SoC) which has been chosen as the reference platform for this work. The Zynq-7000 is a popular heterogeneous platform that includes a dual-core ARM Cortex-A9 processor and a Xilinx 7-Series FPGA fabric. The internal structure of the Zynq-7000 is divided into two main functional blocks: the *processing system* (PS) and the *programmable logic* (PL) [13]. The PS includes the ARM Cortex-A9 processors, interfaces for external memories, a small amount of on-chip RAM memory, and the I/O peripherals. The subsystems in the PS are interconnected among themselves and to the custom logic configured on the PL through an *AMBA AXI* system bus. The main interconnection between the PS and PL consists of a set of memory-mapped AXI (AXI for simplicity) interfaces exported by the PS side to the PL side.

A. System support design

The proposed design is illustrated in Figure 3. In order to support the deployment of dynamically-reconfigured hardware accelerators on the PL, the FPGA area is divided into two main regions: a *static* region and a *reconfigurable* region (denoted by the striped boxes in Figure 3). The static region contains part of the logic that is needed to realize the communication infrastructure, namely a set of AXI Interconnects (discussed in Section IV-A4) and other support modules. Following the specifications of the FRED framework introduced in Section III, the reconfigurable region is organized by following a slotted scheme to host the hardware accelerators.

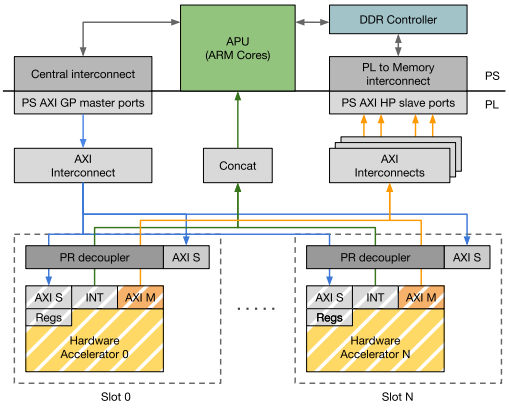

Figure 3. Support design for the Zynq SoC.

To implement the shared-memory communication paradigm reviewed in Section III-B, each hardware accelerator must be able to read and write a memory area that is also accessible from the processors. The Zynq-7000 provides three alternatives for implementing such memory areas: (i) using the internal on-chip memory; (ii) reserving part of the FPGA area to implement a custom memory (using BRAM logic blocks); or (iii) using the main (off-chip) DRAM memory.

Alternative (i) is not viable since the on-chip memory is too small (256 Kb) and hence may not be suitable for supporting the shared-memory communication with multiple HW-tasks. Alternative (ii) determines a waste of the FPGA area, as the synthesis of efficient hardware accelerators generally requires BRAM logic blocks. Conversely, alternative (iii) allows using high-performance AXI interfaces that are directly connected to the DRAM controller. The availability of such interfaces suggests that the Zynq-7000 is prone to support this approach. As a consequence, the main DRAM memory has been selected for implementing the shared-memory paradigm.

Hardware accelerators must be capable of receiving control commands and arguments from the processor and sending synchronization signals to notify their completion. Since each slot of a partition P must be able to host any of the hardware accelerators that implement the HW-tasks with affinity to P, it necessary to define a *common interface*.

1) Common interface: The proposed interface consists of (i) an AXI master interface, (ii) an AXI slave interface exporting a set of control registers and eight 32-bit data registers, and (iii) an interrupt signal. The AXI master interface (denoted as AXI M in Figure 3) has been provided to allow the hardware accelerators to access the DRAM memory through the PS DDR controller. The control registers allow controlling the execution and the state of each hardware accelerator. Data registers can be used for manifold purposes depending on the specific function implemented by the hardware accelerator.

The most common usage consists in storing pointers to memory area in the DRAM (to implement shared-memory communication) or storing control parameters of the HW-tasks. The AXI slave interface (denoted as AXI S in Figure 3) is then used to map the control and data registers into the system memory space, hence making them available from the PS. Finally, the interrupt signal (denoted as INT in Figure 3) is used to notify the completion of the HW-task to the PS.

2) Dynamic partial reconfiguration: In the Zynq-7000 SoC, the FPGA fabric can be fully or partially (re)configured under the control of the software running in the PS using the *device configuration* (DevC) subsystem. Internally, the DevC includes an interface to the *processor configuration access port* (PCAP) and a DMA engine that can be programmed to transfer a bitstream from the main DRAM memory to the PL configuration memory.

Each hardware accelerator corresponds to a bitstream. However, Xilinx tools do not support the relocation of bitstreams [12], i.e., the same bitstream cannot be used to program the same hardware accelerators in different slots. Since FRED requires that a hardware accelerator can be programmed onto different slots (depending on their availability at run-time), it is necessary to synthesize a bitstream for each slot of the partition to which the corresponding HW-task has affinity. Note that this is not relevant for memory consumption, as bitstreams are typically in the order of a few megabytes.

3) Slot decouplers: The reconfiguration process may generate transient glitches that can cause troublesome spurious transactions [12]. To solve this issue each slot is protected by a *partial reconfiguration decoupler* (denoted as PR decoupler in Figure 3), which is used to tie the interface signals to safe logic values. Each decoupler is controlled by the PS by means of a single control register, which is mapped into the memory space using an AXI slave interface.

4) Interconnections: In the proposed design, the AXI master interfaces exported by each slot are connected to a set of

AXI interconnects [14] modules. The proposed connection scheme is based on the rationale of equally distributing the memory bandwidth across the slots using fair arbitration [14]. More articulated connection schemes may be enabled by a fine-grained analysis of the interference incurred by the memory transactions, which is out of the scope of this paper and is left as a future work. The AXI slave interfaces of the hardware accelerators and the decouplers are connected to a single AXI interconnect, which is turn connected to one of the Zynq general purpose master ports (denoted as PS AXI GP in Figure 3). Note that this does not constitute a bottleneck, since the PS is the only master. Finally, the interrupt signals exported by the slots are gathered together in a vector signal and routed to the IRQ_F2P port of the PS.

V. Linux support

This section describes the implementation of the FRED framework for the GNU/Linux operating system. The FRED software support has been designed in a modular fashion relying, as much as possible, on user-space implementation to improve maintainability, safety, and extendability.

The internal architecture of the system is shown in Figure 4. The central component is a user space daemon, named *FRED server*, which is in charge of managing acceleration requests from SW-tasks. The server relies upon two custom kernel modules, and the UIO framework, in order to perform the low-level operations required to control the hardware platform.

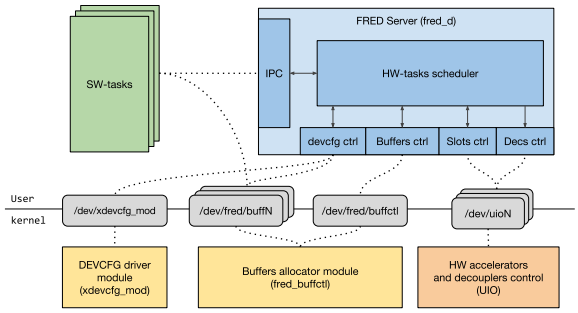

Figure 4. FRED software support architecture.

A. Kernel space

The two aforementioned kernel modules are used to (i) allocate the memory buffers employed to share data between SW and HW-tasks, as described in Section III, and (ii) manage the device reconfiguration. The UIO framework is used for managing hardware accelerators (control and data registers, and interrupt signals) from userspace.

1) Memory Allocator Module: To enforce memory coherence between SW-tasks and HW-tasks, the *shared-memory* infrastructure described in Section III-B, has been implemented using a set of *uncached* memory buffers allocated by a custom kernel module.

The custom kernel module uses the Linux DMA layer to allocate physically contiguous (uncached) memory buffers used to exchange data between HW and SW-tasks. When loaded by the system, the module creates a new character device named `fred_buffctl`, used by the FRED server during the initialization phase to request the allocation of memory buffers.

Each allocation request is performed by an `ioctl` operation and includes, as an argument, the size of the required buffer. 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 userspace.

Once the buffer device has been created, it can be accessed by a SW-task using the Linux standard `mmap()` syscall. When a SW-task calls (from userspace) the `mmap()` on a buffer character device, the corresponding buffer is mapped into its virtual address space. Inside the driver (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 space, it can be accessed by the 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 (note that there are no common cache levels to both the processor and the hardware accelerators, which are directly connected to the DRAM controller). On the other side, a HW-task can access the same buffer through a physical memory address. Such an address is written into the control registers of the HW-task by the FRED server (discussed in details in Section V-B).

In this way, data can be transferred between HW and SW tasks *without any copy operation* or operating system overhead. It is worth observing that under this design the SW-tasks never deal with memory management operations. Each SW-task sees a buffer only as a character device that can be mapped, during its initialization phase, into its virtual memory space. The process of requesting the mapping of such buffers is assisted by a *client support library*.

When the FRED server is shutdown the buffer devices created during the initialization phase are released calling an `ioctl` operation on the `fred_buffctl` device.

2) Reconfiguration Driver: The Zynq FPGA fabric can be reconfigured by the DevC subsystem, as described in Section IV-A2. Under Linux, the DevC is controlled by a kernel driver module designed by Xilinx. Such a driver allocates a character device named `xdevcfg` that can be used to reconfigure the FPGA fabric from the user space, taking a bitstream (introduced in Section IV-A2) as input.

The reconfiguration process is initiated by a `write()` operation on the `xdevcfg` device allocated by the driver. The argument of the write operation is the bitstream file containing the hardware configuration.

The Xilinx's driver has been likely designed with simplicity as a primary design principle. Internally, for each request, the driver allocates a contiguous uncached memory buffer using the `dma_alloc_coherent()` function of the Linux DMA layer. Once the buffer has been allocated and mapped, the driver copies the entire bitstream from the user space to the buffer, using the `copy_from_user()` function of the Linux kernel. Once the bitstream has been copied into the buffer, the driver starts the DevC internal DMA engine for transferring the bitstream from the system memory to the FPGA configuration

memory. After the DMA has been started, the driver performs a busy-wait, polling on a DMA status flag until the transfer has been completed.

This mode of operation is intended to minimize the user efforts to use the driver, but it is clearly unsuitable for the FRED framework because the copy overheads and the busy waits are not compatible with the intensive usage of partial reconfiguration required by the FRED framework. To overcome these issues, the original driver has been modified to take advantage of the allocator module described in the previous section. The rationale is to pre-load all the HW-tasks' bistreams into a set of contiguous memory buffers allocated using the allocator module. Since those operations are performed only once, during the FRED server initialization, they do not produce any overhead at run time.

Once the bitstreams are loaded in physically contiguous memory buffers, they can be reached by the DevC internal DMA engine. For this reason, the driver has been modified to include an `ioctl()` method that allows to start the reconfiguration by passing to the driver a memory reference to a pre-allocated bitstream.

To avoid the busy-wait, the driver has been enhanced with the Linux standard `poll()` method. Once the reconfiguration has been completed, such a method sets the file descriptor of the `xdevcfg` device ready for a read operation. In this way, the reconfiguration process can be easily monitored through POSIX standard I/O multiplexing methods such as `select()` and `poll()`, or the Linux-specific `epoll()`.

With the approach described above, the reconfiguration process is started by an `ioctl()` call on the `xdevcfg` device. The call returns immediately and a user-space application can wait for the end of the reconfiguration without busy-waiting.

B. User space

The FRED server is the main userspace component of the FRED software support. Form an architectural perspective it is organized as an event-driven system. Internally, the server includes a core component, named "HW-Tasks scheduler", supported by a layer of software libraries used to perform low-level operations, as shown in Figure 4. Conceptually, the FRED server interacts with the rest of the system by means of two main software interfaces, one dedicated to interprocess communications with SW-tasks and the other used to interact with the low-level support.

During the initialization phase the FRED server reads two configuration files containing the description of the hardware design. 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 such files, the FRED server initializes its own data structures and requests the allocator module to allocate the memory buffers used for both bitstreams and data sharing.

1) Communication mechanism: The communications channels between the FRED server and SW-tasks rely upon *Unix domain sockets*. After the initialization phase, the server instantiates a listening socket, named `fred_sock`, used by SW-tasks to establish a new connection. Once the connection is established, the SW-task can send requests to the server.

To make the system more usable from a client programmer perspective, communication functions between SW-tasks and the FRED server are encapsulated into the client support library. The communication pattern between FRED server and a SW-task is presented in Figure 5. Once the server setup is completed, a SW-task can initiate a new connection by calling the `fred_init_hwt()` function (see Figure 5). The FRED server replies back with a message containing the buffers description in terms of device files and sizes. Using this data the SW-task can map the buffers into its own address space. Again, such a mapping operation is assisted by the client support library.

At this point, the SW-task can fill the input buffers of its associated HW-task by simply writing into the corresponding memory locations without any additional overhead. Once input data have been prepared, the SW-Task can request the execution of its HW-task by calling the `fred_exec_hwt()` function. This function call causes the SW-task to be suspended until the completion of the HW-task. When the HW-Task completes, the SW-Task resumes its execution and can retrieve the data from the output buffers.

It is worth noticing that SW-tasks never interact directly with the hardware, nor they are required to perform privileged operations. Any interaction between client SW-tasks and the platform hardware are mediated by the FRED server.

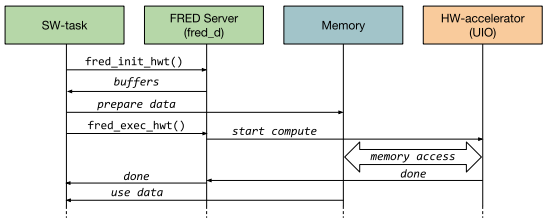

Figure 5. Communication between SW-Tasks, FRED server, and HW.

2) Event loop: The main component of the HW-tasks scheduler is a state machine driven by an event loop. The event loop monitors the file descriptors exported by the low-level and communication interfaces and drives the state machine to handle incoming client requests and hardware events. Internally, the event loop is built around the `epoll` system call. The classes of events handled by the event loop are: (i) Completion of the device configuration; (ii) Completion of a HW-Task; (iii) Connection request from a SW-Task; (iv) Message from a SW-Task.

VI. Experimental Results

This section describes a set of experiments aimed at evaluating the performance of the FRED software support. The experiments have been carried out on the Digilent's Zybo board featuring the Zynq-7010 SoC and running Xilinx's PetaLinux.

A. Performance Evaluation of the Reconfiguration Driver

The first experiment evaluated the improvements achieved using the customized device reconfiguration driver with respect to the Xilinx driver.

Measurements were done by running a single dummy task triggering 10^6 reconfiguration requests to the driver, each of them configuring a 338 KB bitstream into an FPGA slot. During the experiment the system was not loaded, hence the DevC

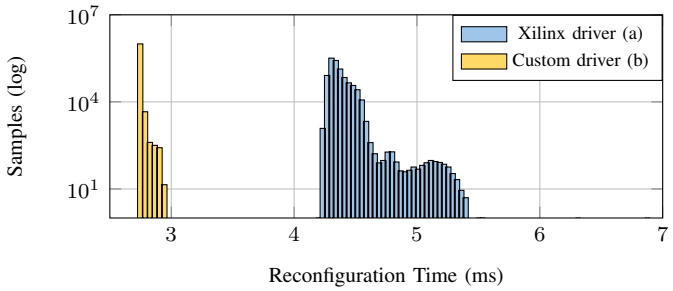

Figure 6. Distributions of the reconfiguration times.

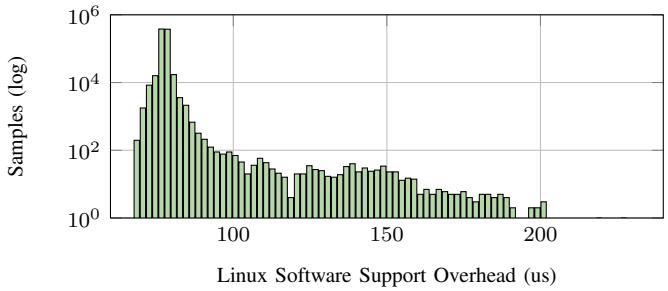

Figure 7. Distribution of the overhead introduced by FRED.

DMA did not suffer from any interference on the system bus while reading the bitstream from memory.

Figure 6 shows the reconfiguration times measured when using the original Xilinx driver, and those obtained with the custom driver developed in this work. While the average reconfiguration time using the Xilinx driver is $4.340\ ms$, the custom one reduces the average reconfiguration time to $2.755\ ms$, with an approximate speedup of 1.574. Moreover, the worst-case reconfiguration time measured for the original driver is $6.876\ ms$, improved with a speedup of 2.340 by the custom driver, for which the longest measured reconfiguration time was $2.940\ ms$.

These results shows that our approach improves the reconfiguration time while decreasing the variance from $4.48 \cdot 10^{-3}$ (for the Xilinx driver) to $1.62 \cdot 10^{-5}$ (for the custom driver).

B. Overhead of the Linux Support

The second experiment was aimed at measuring the overhead introduced by the FRED software support while serving the requests generated by the SW-tasks.

To evaluate the net overhead introduced by the infrastructure (namely inter-process communication and the management of hardware events), the experiment was performed with a basic system configuration consisting of a single SW-task running in the system, requesting for hardware accelerations to the FRED server. More specifically, the experiment measured the overheads introduced by the FRED software support when calling `fred_exec_hwt()`, that includes:

- The time elapsed from the acceleration request to the instant at which the FRED server triggers the driver to perform the hardware reconfiguration;
- The interval from the time at which the driver notifies the FRED server with the end of reconfiguration to the time at which the FRED server starts the HW-task;
- The interval between the time at which the HW-Task notifies the FRED server its completion and the time at which the SW-Task is awakened.

Figure 7 reports the distribution of the sums of the aforementioned latencies measured for each acceleration request performed by the SW-Task. The longest measured overhead resulted $227.125\ \mu s$, while the average delay was $77.978\ \mu s$. Please note that the overhead introduced by the FRED software support does not depend upon the amount of data shared between SW and HW.

VII. Conclusions

This paper presented the design and implementation of the FRED framework on the Linux operating system to ease the exploitation of FPGA accelerators in real-time applications running on the Zynq-7000 platform. The software includes a kernel module for implementing shared-memory communication with hardware accelerators, an improved driver to handle the FPGA reconfigurations, and a user-space daemon to schedule the requests of hardware acceleration.

Experimental results showed that the proposed approach allows halving the reconfiguration times with respect to the official driver released by Xilinx, with a speedup of 2.340. It has also been shown that the features offered by the FRED software support introduce an overhead that can be tolerated by several applications, with a maximum measured overhead less than $228\ \mu s$. Future work includes the analysis of the delays incurred by the hardware accelerators when accessing the AXI bus and the memory controller.

References

[1] A. Biondi, A. Balsini, M. Pagani, E. Rossi, M. Marinoni, and G. Buttazzo, "A framework for supporting real-time applications on dynamic reconfigurable fpgas," in *Proc. of the IEEE Real-Time Systems Symposium (RTSS 2016)*, December 2016, pp. 1–12.

[2] K. Danne and M. Platzner, "Periodic real-time scheduling for fpga computers," in *Proc. of the 3rd Int. Workshop on Intelligent Solutions in Embedded System*, May 2005.

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

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

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

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

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

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

[9] R. T. E. Ltd. Freertos real-time operating system. [Online]. Available: http://www.freertos.org/

[10] H. K.-H. So and R. Brodersen, "A unified hardware/software runtime environment for fpga-based reconfigurable computers using borph," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 2, pp. 14:1–14:28, January 2008.

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

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

[13] *Zynq-7000 AP SoC Technical Reference Manual*, Xilinx, 2015.

[14] *AXI Interconnect, LogiCORE IP Product Guide*, Xilinx, 2016.