Enabling Predictable Hardware Acceleration in Heterogeneous SoC-FPGA Computing Platforms

Marco Pagani

Scuola Superiore Sant'Anna
Université de Lille
École doctorale des sciences pour l'ingénieur

Thèse de doctorat en informatique et applications préparée au sein des laboratoires ReTiS et CRIStAL

Enabling Predictable Hardware Acceleration in Heterogeneous SoC-FPGA Computing Platforms

Techniques pour l'amélioration de la prévisibilité de l'accélération matérielle pour les plateformes informatiques hétérogènes SoC-FPGA

Marco Pagani

Supervisors - Directeurs de thèse
Giorgio Buttazzo, Giuseppe Lipari

Tutor
Mauro Marinoni

Soutenue le 20 juillet 2020 devant le jury composé de:

Smail Niar	Université Polytechnique Hauts-de-France	Examinateur
Jörg Henkel	Karlsruher Institut für Technologie	Examinateur
Luca Abeni	Scuola Superiore Sant'Anna	Examinateur
Francesca Palumbo	Università degli Studi di Sassari	Examinatrice
Sanjoy Baruah	Washington University in St. Louis	Rapporteur
Liliana Cucu-Grosjean	INRIA de Paris	Rapportrice
Giusppe Lipari	Université de Lille	Directeur
Giorgio Buttazzo	Scuola Superiore Sant'Anna	Directeur

Ph.D. course in emerging digital technologies
Academic year 2018/2019

Abstract

Modern computing platforms for embedded systems are evolving towards heterogeneous architectures comprising different types of processing elements and accelerators. Such an evolution is driven by the steady increasing computational demand required by modern cyber-physical systems. These systems need to acquire large amounts of data from multiple sensors and process them for performing the required control and monitoring tasks. These requirements translate into the need to execute complex computing workloads such as machine learning, encryption, and advanced signal processing algorithms, within the timing constraints imposed by the physical world. Heterogeneous systems can meet this computational demand with a high level of energy efficiency by distributing the computational workload among the different processing elements.

This thesis contributes to the development of system support for real-time systems on heterogeneous platforms by presenting novel methodologies and techniques for enabling predictable hardware acceleration on SoC-FPGA platforms. The first part of this thesis presents a framework designed for supporting the development of real-time applications on SoC-FPGAs, leveraging hardware acceleration and logic resource "virtualization" through dynamic partial reconfiguration. The proposed framework is based on a device model that matches the capabilities of modern SoC-FPGA devices, and it is centered around a custom scheduling infrastructure designed to guarantee bounded response times. This characteristic is crucial for making dynamic hardware acceleration viable for safety-critical applications. The second part of this thesis presents a full implementation of the proposed framework on Linux. Such implementation allows developing predictable applications leveraging the large number of software systems available on GNU/Linux while relying on dynamic FPGA-based hardware acceleration for performing heavy computations. Finally, the last part of this thesis introduces a reservation mechanism for the AMBA AXI bus aimed at improving the predictability of hardware accelerators by regulating BUS contention through a bandwidth reservation mechanism.

Riassunto

Le moderne architetture di calcolo per sistemi integrati sono in via di evoluzione verso piattaforme sempre più eterogenee, comprendenti diverse tipologie di processori e acceleratori. Tale evoluzione è guidata dalla necessità di soddisfare la crescente richiesta di capacità di calcolo da parte dai moderni sistemi cyber-fisici. Questi sistemi hanno la necessità di acquisire e processare grandi quantità di dati, provenienti da differenti sensori, in modo da poter eseguire le necessarie operazioni di controllo e supervisione. Tali requisiti si traducono nella necessità di eseguire carichi computazionali complessi quali algoritmi per l'apprendimento automatico, l'elaborazione numerica dei segnali e la crittografia, nel rispetto dei vincoli temporali imposti dall'interazione con il mondo fisico. Le piattaforme eterogenee consentono di soddisfare questa domanda di calcolo distribuendo il carico di lavoro di calcolo tra i diversi processori e acceleratori, mantenendo quindi un elevato livello di efficienza energetica.

Questa tesi contribuisce allo sviluppo del supporto per i sistemi in tempo reale su le piattaforme eterogenee, presentando un insieme di tecniche e metodologie per rendere predicibile l'accelerazione hardware su piattaforme SoC-FPGA. La prima parte di questa tesi presenta un *framework* progettato per supportare lo sviluppo di applicazioni in tempo reale su piattaforme SoC-FPGA, sfruttando l'accelerazione hardware e la riconfigurazione dinamica parziale per «virtualizzare» le risorse logiche. Il *framework* è basato su un *device model* che esprime le caratteristiche delle moderne piattaforme SoC-FPGA, ed è strutturato intorno a un'infrastruttura di schedulazione progettata per garantire tempi di risposta limitati. Questa caratteristica è fondamentale per rendere l'accelerazione hardware dinamica utilizzabile nel contesto dei sistemi critici. La seconda parte della tesi presenta un implementazione completa del *framework* proposto su Linux. Grazie a questa implementazione, è possibile sviluppare applicazioni predicibili in ambiente GNU/Linux, sfruttando l'accelerazione dinamica basata su FPGA per eseguire le operazioni computazionalmente più pesanti, utilizzando al contempo la grande quantità di software e librerie offerte dall'ambiente. Successiva-

mente, l'ultima parte di questa tesi presenta un meccanismo di regolazione della banda per il bus AMBA AXI concepito per migliorare la predicibilità dell'accelerazione hardware sulle piattaforme eterogenee.

Résumé

Les architectures informatiques modernes pour les systèmes intégrés évoluent vers des plateformes de plus en plus hétérogènes, comprenant différents types de processeurs et d'accélérateurs. Cette évolution est entraînée par la nécessité de répondre à la demande croissante de capacité de calcul par les systèmes cyber-physiques modernes. Ces systèmes doivent acquérir et traiter de grandes quantités de données, provenant de différents capteurs, afin d'exécuter les tâches de contrôle et de surveillance nécessaires. Ces exigences se traduisent par la nécessité d'exécuter des charges de calcul complexes telles que des algorithmes d'apprentissage automatique, de traitement numérique des signaux et de cryptographie, en respectant les contraintes de temps imposées par l'interaction avec le monde physique. Les plateformes hétérogènes permettent de répondre à cette demande de calcul, en distribuant le travail entre les différents processeurs et accélérateurs, ce qui permet de maintenir un niveau élevé d'efficacité énergétique.

Cette thèse contribue au développement du support pour les systèmes temps réels sur des plateformes hétérogènes, en présentant un ensemble de techniques et de méthodologies pour rendre prévisible l'accélération matérielle sur les plateformes SoC-FPGA. La première partie de cette thèse présente un *framework* pour soutenir le développement d'applications en temps réel sur les plateformes SoC-FPGA, en utilisant l'accélération matérielle et la reconfiguration dynamique partielle pour « virtualiser » les ressources logiques. Le framework est basé sur un *device model* qui exprime les caractéristiques des plateformes SoC-FPGA modernes, et il est structuré autour d'une infrastructure d'ordonnancement conçue pour garantir des temps de réponse limités. Cette caractéristique est fondamentale pour rendre l'accélération matérielle dynamique viable dans le contexte des systèmes critiques. La deuxième partie de la thèse présente une implémentation complète du *framework* proposé sur Linux. Grâce à cette implémentation, il est possible de développer des applications prévisibles dans l'environnement GNU/Linux, en profitant de l'accélération dynamique basée sur FPGA pour exécuter les opérations de calcul les plus intensifs, tout en utilisant la grande quantité de logiciels et de

bibliothèques offerts par l'environnement. Ensuite, la dernière partie de cette thèse présente un mécanisme de régulation de la bande pour le bus AMBA AXI conçu pour améliorer la prévisibilité de l'accélération matérielle sur les plateformes hétérogènes.

Contents

Chapter 1

Introduction

This chapter presents the motivations behind the development of this thesis. First, an overview of traditional software-programmable architectures like general-purpose processors and graphic processing units is presented, showing their advantages and limitations. Then, the opportunity of using FPGA platforms as computing devices for developing custom hardware accelerators are discussed and confronted with the existing constraints. Later, the advantages offered by modern heterogenous SoC-FPGAs are discussed, describing how these platforms can be leveraged to combine the advantages of traditional software-programmable architectures with the benefits offered by reconfigurable FPGA-based acceleration. Finally, the challenges arising from leveraging FPGA-based computing to accelerate real-time applications are discussed, presenting the solution proposed in this thesis to address these challenges.

1.1 FPGA as computing devices

FPGA devices have been traditionally employed for the rapid prototyping of digital logic and for low-volume applications whose limited production numbers do not justify the manufacturing of dedicated application-specific integrated circuits (ASIC) chips. However, recently, there has been a renewed interest in FPGAs as platforms for implementing hardware accelerations. The steady increase in the computational demand required by modern applications, from cyber-physical systems to the cloud services, began to show the limits of traditional general-purpose software-programmable architectures, such as CPUs and current GPUs, in terms of energy efficiency. By leveraging the FPGA programmable fabric, it is possible to deploy custom hardware accelerators with datapaths tailored for the specific algorithms required by the application. In this way, it is possible to support the execution of complex computing workloads, such as machine learning algorithms and image/video processing, within the timing and energy constraints imposed by physical and performance requirements.

1.1.1 Software programmable devices

Traditional programmable general-processors are based on the Von Neuman architecture. In this computing paradigm, a processor constituted by a control unit and one or more execution units. The processor is connected to a memory containing both data and instructions. In the most simple case, the execution unit comprises a single functional unit, the arithmetic logic unit (ALU). In this architecture, the processor cyclically (i) fetches and decodes the current instruction (pointed by a program counter register) from memory, (ii) fetches the operands, (iii) processes them using the execution unit, (iv) write the results to a register file or memory, and (v) update the program counter. Although modern CPUs are remarkably complex devices, capable of concurrently executing multiple instructions out of order using several functional units, they are still conceptually based on this model architecture. This main limitation of the Von Neuman architecture comes from the fact that it is intrinsically limited by the rate at which instructions and data can be retrieved and written from memory. Modern state-of-the-art CPUs mitigate this issue in many ways, like using a Harvard architecture with large separate caches for instructions and data or leveraging instruction-level parallelism to execute multiple instructions simultaneously. However, this fundamental limitation still exists, and many research and industrial efforts are dedicated to finding efficient mitigation strategies.

1.1.2 SIMD processing and GPUs

Many real-world computationally intensive applications, like video and image processing, signal processing, etc. need to perform the same sequence of operations over large sets of data. Hence, one possible and yet very successful strategy to improve the performances of general-purpose processors has been exploiting data parallelism by designing functional units capable of simultaneously performing the same operation on multiple data operands. This paradigm is referred to as single instruction multiple data (SIMD) in Flynn's taxonomy. In this way, the fetch and decoding stages of a single instruction correspond to many parallel operations on multiple data. Historically, the SIMD paradigm has been first implemented in high-end vector processors. Afterward, many vendors augmented their off-the-shelf general-purpose processors with dedicated SIMD units accessible through dedicated instruction set extensions. Finally, this paradigm has been adopted by modern general-purpose GPU, which are essentially multicore architectures consisting of many GPU cores. Internally, each GPU core has multiple SIMD lanes capable of parallel data processing.

1.1.3 Instruction set architecture

Computer programmers can write programs relying on an abstract model of the processor called instruction set architecture (ISA). The ISA is a detailed functional specification of the processor describing how the instructions operate on the registers or the memory, and other aspects of the processor's behaviors. The same ISA can be implemented in different ways depending on the specific requirements in terms of performances, costs, and energy. Concrete implementations of an ISA realized using digital logic are referred to as microarchitectures. The ISA provides a compatibility layer that is a fundamental abstraction in software computing. In fact, a program built for an ISA can indeed run on all processors implementing the same ISA. Current software-programmable processors, including CPUs and GPUs, are programmed using the machine instructions specified by their ISAs. Hence, their microarchitectures are designed to conform to the architectural model detailed by the ISA. Such microarchitectures are thus centered around a general-purpose programmable datapath built of functional units that can be configured by the control unit according to a sequence of instructions.

1.1.4 FPGA as programmable computing devices

FPGAs are radically different computing devices with respect to software-programmable processors in the sense that they do not rely on a predefined microarchitecture or an ISA to be programmed. Instead, "programming" an FPGA means designing the datapath itself that will be used to process the data. The resulting design will be implemented as digital logic using

the FPGA's logic cells. In a certain sense, FPGAs are one level of abstraction below software-programmable processors providing the possibility to design the computing microarchitecture itself, as illustrated in Figure 1.1. An FPGA configuration does not contain a program, in the form of a sequence of instructions, but rather the set of state information required to configure the programmable fabric to form the specified logic. This hardware-programmability means that FPGAs can be used in very flexible ways. For instance, an FPGA can be configured for implementing a custom hardware accelerator designed with a processing datapath tailored for efficiently implementing in hardware a specific algorithm. Still, it can also be configured to form the logic of a software-programmable microarchitecture implementing the ISA of a processor.

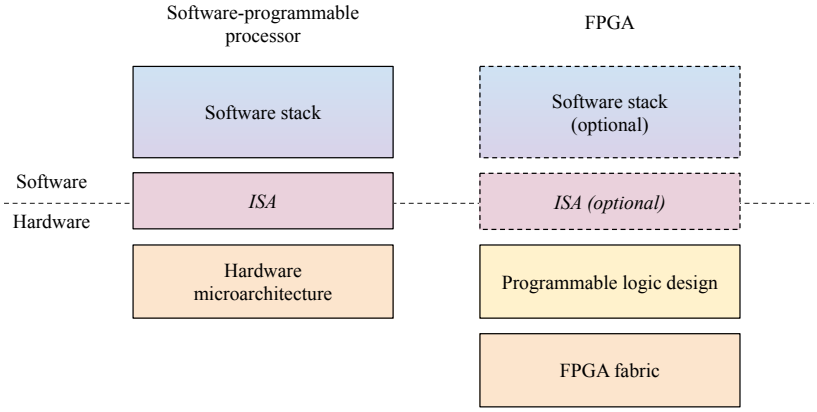

Figure 1.1: Comparison of the abstraction layers in a software-programmable processor and in an FPGA-based system.

Figure 1.2 illustrates how the inference process for a deep artificial neural network (a) can be performed considering different computing paradigms. With a traditional general-purpose processor (b), the computations required by each layer are executed sequentially. SIMD/vector units (c), like the one present in GPU cores, can speed up the computation by performing multiple operations in parallel, exploiting data parallelism. However, each inference run still needs multiple cycles to be completed. By leveraging the FPGA fabric, it possible to implement the inference process with a custom architecture (d) in which the computations required by each layer are performed by custom functional units (FU). With this approach, the throughput of the interference process can be increased without the need for ramping up the clock frequency.

Traditionally, this "hardware programming" has been carried out using hardware description languages (HDL) like VHDL or Verilog, which are based on a different paradigm with respect to software programming languages such as C or Python. At a very high level, software programming languages allow

describing an algorithm in terms of a sequence of statements that will be sequentially executed by a processor or interpreted by an interpreter. On the other hand, hardware description languages allow specifying sets of operations that will be performed concurrently when the activation conditions are met, eventually triggering other operations. This coding model allows the HDL language synthesizer to transform an HDL description into a digital logic implementation consisting of multiple independent entities (such as adder, multiplexers, and flip-flops) that operate concurrently. Such a digital logic implementation can then be mapped onto the logical primitives of an FPGA.

However, the inherent complexity of hardware design and debugging, together with the lack of knowledge on hardware description languages among software programmers, have traditionally relegated FPGA-based computing to niche applications where the benefits outnumber the additional development costs.

1.1.5 High-level sysnthesis

Many research and industrial efforts have been dedicated to bridging the gap between software programming languages for general-purpose processors and hardware programming languages. These efforts culminated in the development of high-level synthesis (HLS) tools, which allow translating an algorithmic description of a procedure into a hardware implementation [46]. The algorithmic description provided as an input to an HLS tool is typically coded using standard software programming languages, such as C or C++, extended with specific statements or compiler directives (pragmas) to optimize the code for hardware implementation. These extensions are often crucial for improving the HLS translation process helping the tool generating an efficient hardware description. At the end of the translation process, HLS tools typically generate a hardware implementation in the form of a register-transfer level (RTL) description of a processing datapath controlled by a control unit consisting of a set of finite-state machines.

HLS tools are crucial for enabling software programmers to access FPGA-based computing and improving the productivity of digital hardware design in general. Although hardware designs generated using current HLS tools tend to be less efficient with respect to native HDL designs, the huge leaps in productivity provided by a HLS-based design flow often offset these implementation inefficiencies.

1.1.6 FPGA-based hardware accelerators

As previously described, FPGAs are based on a hardware-programmable fabric that can be configured to implement any kind of digital processing logic, from software-programmable processors to dedicated hardware accelerators. This thesis addresses the specific case in which an FPGA is used for acceler-

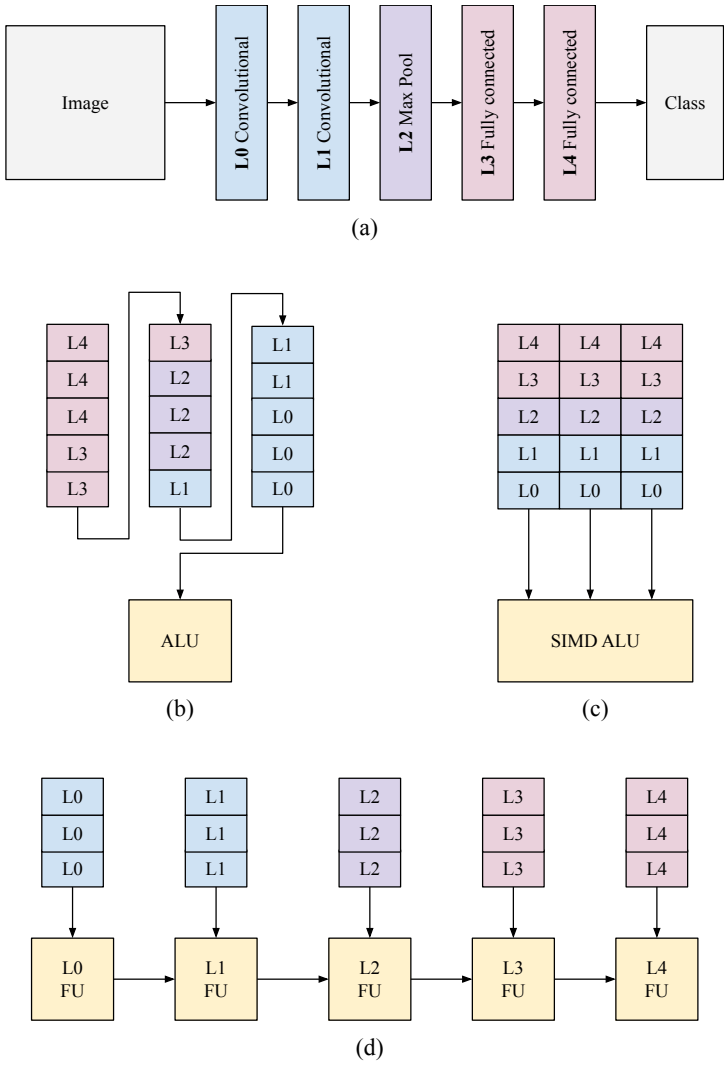

Figure 1.2: Illustration of a deep neural network (a) inference process performed using different computing paradigms. (b) traditional general-purpose processor, (c) SIMD/vector unit like the one present in GPU cores, (d) dataflow processing on FPGA (d) using custom functional units (FU) for each layer.

ating real-time applications using custom hardware accelerators generated using a HLS-based flow. Many real-time applications consist of cyclic and sporadic activities. These activities often include i) a control logic portion, typically implemented using conditional logic statements such as *if–then–else* chains, and ii) computationally intensive portions that may present some level of data-parallelism. These latter computationally intensive parts are prone to be accelerated using some form of parallel computing hardware. In modern systems, these sections are often accelerated using high-performance programmable GPUs. Hence, these computationally intensive code sections are implemented using sequences of compute kernels. These kernels are then compiled for the specific GPU's architecture and later submitted at run-time to the GPU cores to be executed.

In modern design flows, compute kernels are coded using dedicated frameworks such as OpenCL or CUDA, which are based on a programming model that is somehow similar to the ones used in traditional software programming. Moreover, the languages used for describing these kernels are based on popular system programming languages such as C or C++. While this approach is clearly convenient in terms of code productivity, it also comes with its intrinsic limitations since GPUs are still software computing devices with a programmable datapath. On the one hand, this software programmability offers a crucial abstraction that allows these devices to be conveniently programmed with a predefined instruction set. On the other hand, it comes with its intrinsic limitations in terms of processing efficiency since, to be general-purpose programmable, the architecture has to be oriented to a wide range of computations rather than being tailored for a specific class of algorithms. While software programming optimization techniques can mitigate some of these structural constraints, they cannot circumvent fundamental architectural limitations.

Conversely, by leveraging FPGA-based acceleration, it is possible to rule out the limitations imposed by software-programmable architectures. The FPGA programmable logic fabric allows implementing custom accelerators with datapaths tailored for the specific code portions to be accelerated. These accelerators do not need to implement a pre-existing ISA. Hence, they are not architecturally constrained to a specific processing paradigm. As a result, they can deliver high processing throughput while being energy-efficient. In other words, by skipping the software-programmability abstraction layer, it is possible to address the root limitations of traditional programmable architectures. For instance, Figure 1.3 shows a simple application consisting of two cyclic activities, A and B, making use of dedicated FPGA-based hardware accelerators to speed up their computations. Each job of activity A calls hardware accelerator A to speedup its computationally intensive portions. Likewise, each job of activity B relies on hardware accelerator B to speedup its computationally intensive portions.

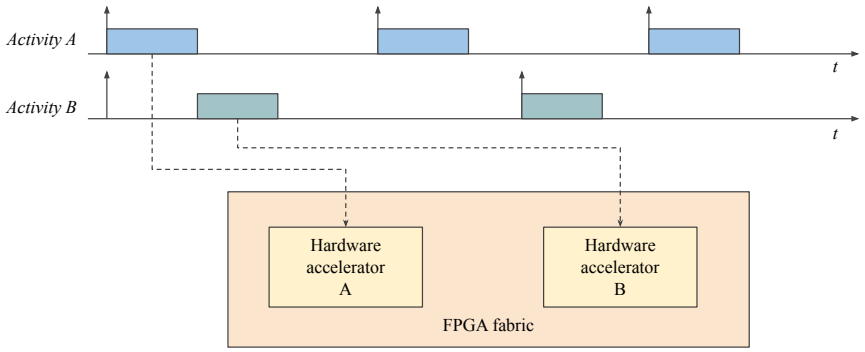

Figure 1.3: Illustration of a real-time application making use of FPGA-based hardware acceleration.

1.1.7 Overheads and limitations of HW-programmability

While FPGA's hardware-programmability offers unique flexibility, it also comes at a cost. When compared to a standard-cell implementation, configurable logic cells and routing resources introduces significant overhead in terms of area requirements. Considering the same digital logic circuit, an FPGA implementation would require, on average, 20 times more silicon area, and it would result in 3 to 4 times slower in terms of clock cycles compared to a standard-cells ASIC implementation [38]. Moreover, FPGA implementations consume more dynamic power with respect to ASIC implementations. While the exact measure of this gap depends on the current technological node, it is very like that this overhead will continue to be present even for future generations of FPGA fabric. Nevertheless, modern FPGA tries to bridge this gap using special-purpose logic cells that can improve the efficiency by hardwiring selected primitive functions directly on silicon (hard blocks).

1.2 Predictability of hardware accelerators

One of the most fundamental properties that real-time systems should have is predictability. In order to be predictable, a system must be itself composed of predictable and well-documented components that do not present unexpected behaviors. When a system is predictable, it becomes feasible to derive a response time analysis providing worst-case response time bounds of the system's activities. On the contrary, if the system includes black-box components, documented only with limited and vague information, it can become very challenging or even impossible to derive reasonable worst-case response time bounds. In the context of cyber-physical systems, these response-time bounds are crucial for guaranteeing that the system can safely operate within the timing constraints imposed by the physical environment.

In this respect, the adoption of GPU-based accelerations for real-time

systems may pose some serious concerns. Currently, only minimal and nebulous information on the internal architecture and resource management logic is publicly available for state-the-art GPUs. Moreover, modern GPUs are usually programmed using very complex software stacks, often including proprietary software components. These software stacks add another layer of uncertainty, which further complicates the development of a realistic system model required to derive safe worst-case response time bounds. On the contrary, FPGA-based hardware accelerators are intrinsically more open, offering the possibility to simulate and even describe their internal architectures at clock-level granularity. Hence, FPGA-based accelerators can be modeled with higher precision, providing the system designer with accurate estimations of the worst-case response times [19, 51]. These characteristics have made FPGA-based acceleration attractive in several safety-critical domains such as signal processing, machine learning algorithms, convolutional neural networks, and many other computationally-intensive applications [31, 34, 72].

Another benefit of FPGA acceleration emerges when considering that practical hardware accelerators need to access shared resources such as the system bus for exchanging data with the rest of the system. Typically, FPGA-based hardware accelerators are connected to the remainder of the system through an interconnect bus. Such a interconnect represents a contention point since it is concurrently accessed by the accelerators for reaching the data allocated in the main system memory. In this respect, the FPGA-based approach to hardware acceleration provides some advantages since portions of the interconnect itself are implemented on the programmable FPGA fabric. Hence, they can be customized by attaching dedicated bus modules such as bandwidth regulators or even replaced with custom interconnects modules implementing the desired arbitration policy.

1.3 Heterogenous SoC-FPGAs

As previously discussed, many real-time applications consist of cyclic and sporadic activities comprising control logic portions often interleaved with computationally intensive portions. On the one hand, computationally intensive portions usually present some form of data-parallelism that can be exploited using parallel hardware. On the other hand, control logic portions comprise conditional statements that present control and data dependencies. These statements need to be evaluated sequentially and hence are not suitable for parallel execution. For this reason, control sections are prone to be coded as software processes to be executed on a general-purpose processor.

This kind of real-time application can be implemented on traditional FPGA chips leveraging the versatility of the programmable fabric. Software activities can be executed on a soft-core processor, which is a general-purpose software-programmable processor implemented using the FPGA fabric logic

cells. At the same time, the computationally intensive section can be coded and designed as custom hardware accelerators managed by the soft-core processor. However, this type of implementation has its intrinsic drawbacks. Soft-cores provides significantly worse performances when compared to hard processors implemented directly on silicon. In recent years, SoC-FPGAs platforms have emerged as an attempt to tackle this limitation by leveraging the hardware-programmability of the FPGA fabric while retaining the advantages of traditional general-programmable processors.

SoC-FPGAs are heterogeneous platforms comprising a system-on-chip device tightly coupled with FPGA hardware-programmable fabric, as shown in Figure 1.4. The SoC side typically includes traditional SoC functional components such as i) one or more general-purpose cores, ii) a memory controller, iii) a set of internal peripherals providing I/O capabilities, and iv) an interconnect connecting all components. SoC-FPGAs are extendable devices in the sense that the FPGA fabric can be used to implement additional hardware modules for extending the functionalities of the SoC. In modern Xilinx's SoC-FPGA, the SoC part of the device is called the processing system (PS), while the FPGA side is referred to as programmable logic (PL). In Intel (previously Altera) devices, the SoC side is referred to as hard processor system (HPS). The general-purpose hard processor cores included in the SoC side offers a huge performance improvement with respect to soft-cores. Moreover, they can be used without consuming valuable programmable resources of the FPGA fabric for implementing a soft-core.

Real-time applications can be efficiently deployed on these SoC-FPGA heterogeneous platforms by executing the control logic portion on the general-purpose hard cores while implementing the computationally intensive portions as hardware accelerators deployed on the FPGA.

1.4 Reconfigurable hardware

The vast majority of modern FPGAs and SoC-FPGAs support dynamic partial reconfiguration (DPR). This feature allows reconfiguring at run-time a subset of the FPGA's logic resources while the remaining resources continue to operate without being interrupted. By leveraging DPR, it is possible to share one or more portions of FPGA fabric between multiple hardware modules in time-multiplexing. This capability is especially interesting when considering that real-time applications typically consist of periodic or sporadic activities. In this context, implementing the computationally intensive portions of the activities as statistically allocated hardware accelerators can result in an underutilization of the FPGA fabric since logic resources may idling most of the time. However, by leveraging DPR, it is possible to reconfigure the fabric on the fly, implementing the required hardware accelerators only when they are actually needed by a software activity for performing the necessary

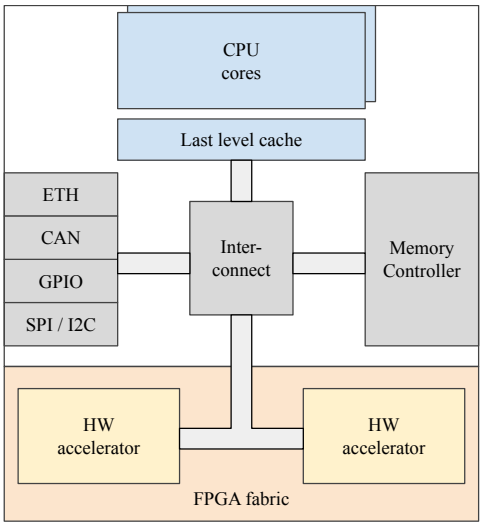

Figure 1.4: Block diagram of a SoC-FPGA platform.

computations.

From a real-time systems perspective, this scenario opens an entirely new scheduling dimension for applications on FPGA platforms. The traditional concept of multitasking can be extended to hardware activities in the form of hardware accelerators. Similarly to software multitasking, where multiple tasks share the processors by switching contexts, DPR allows interleaving multiple hardware activities on the FPGA fabric. In this way, it is possible to improve logic resource utilization. Moreover, similarly to the abstraction of a virtual processor, DPR can be used to provide the abstraction of a "virtual FPGA capable" of hosting more hardware activities than what is statically possible within its physical resources.

While this paradigm presents many opportunities, it also poses multiple open issues. Some of these queries arise from current FPGA's architectural limitations, such as the limited reconfiguration rate and the impossibility of reconfiguring more than one portion of FPGA at a time. Some others are more fundamental such as which kind of scheduling policy is suitable for managing the reconfiguration and the execution of the hardware accelerators on FPGA. Moreover, the contention costs of concurrent accesses to shared resources such as system bus must be considered, and the accesses must be supervised with a suitable mechanism for preventing overruns.

This thesis addresses these issues by presenting a novel approach for supporting the development and execution of real-time applications on SoC-FPGA platforms. First, Chapter 2 proposes an essential background on the enabling technologies used in this thesis. Then, Chapter 3 presents a framework designed for supporting predictable hardware acceleration of real-time applications using dynamic partial reconfiguration to "virtualize" the FPGA

fabric resources. With this approach, it is possible to host more hardware accelerators in time-sharing than could otherwise be statically allocated on the physical fabric. Later, Chapter 4 proposes a full implementation of the proposed framework on GNU/Linux, which allows leveraging the benefits of FPGA-based predictable acceleration while relying on the available software libraries and services available on a rich operating system. Finally, Chapter 5 introduces a bus bandwidth regulation and reservation mechanism aimed at enhancing system safety and predictability by regulating the access to shared resources.

Chapter 2

Essential background

This chapter presents an essential background on the enabling technologies behind heterogenous SoC-FPGAs that are the reference platforms used for the development of this thesis. First, FPGA devices are introduced as reconfigurable platforms for implementing logic functions, and then their internal structure is briefly discussed. Later, the design flow used for implementing digital logic on the FPGA fabric is discussed. Finally, an overview of the partial reconfiguration mechanism is presented.

2.1 Overview of FPGAs architecture

FPGAs are integrated circuits designed to be configured after manufacturing for implementing arbitrary logic functions in hardware. In this sense, they differ from ASICs, which are custom manufactured to implement a set of specific functionalities fixed at design time. The process of customizing the FPGA fabric for implementing a specific logic function is referred to as device configuration. An FPGA configuration, unlike a CPU program, does not specify a set of operations that will be interpreted and executed by the FPGA, but rather contains the actual configuration data needed to set its internal components for implementing the desired logic function in the form of a digital circuit.

2.1.1 Internal architecture

In ASICs chips, logic functions are implemented by hard-wiring together logic gates at manufacturing time. On FPGAs, by contrast, logic functions are implemented by means of configurable logic cells. These configurable logic elements are typically built around a look-up table (LUT).

Look-up tables

A *n-inputs* LUT is a logic circuit that can be configured to implement any possible combinational logic function with n inputs and a single output. For a LUT with n inputs, 2^n configuration memory cells are needed to store the truth table of the logic function to be implemented. Then, a $2^n : 1$ multiplexer is used to route the state bit from one of the configuration memory cells to the output according to the inputs values. Figure 2.1 illustrates a simplified 2-input look-up table implementing an AND gate.

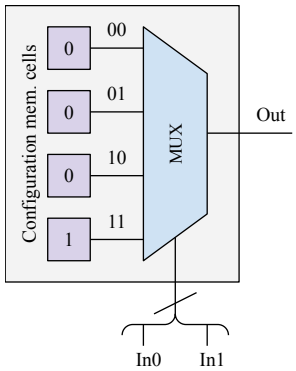

Figure 2.1: Illustration of a simplified 2-input look-up table.

Configuration cells

At the physical level, configuration memory cells can be implemented using different technologies. Depending on the need for constant power in order to retain data, these technologies be classified into two main categories (i) volatile storage technologies, like static RAM (SRAM) cells, and (ii) non-volatile storage technologies like flash memory or antifuse cells. Leading FPGA manufacturers, like Xilinx and Intel (previously Altera), use SRAM-based configuration memory cells, since this technology provides high logic densities. Moreover, unlike flash memory, SRAM cells do not wear out while being written. This characteristic is crucial in the context of this thesis since with the proposed approach for FPGA-based hardware acceleration the fabric needs to be continuously configured at run-time. Hence, this work considers only FPGAs with SRAM-based configuration memory.

Logic cells

Elementary logic cells are built combining one or more look-up tables with flip-flops (FF) capable of storing a state. This allows logic cells to implement sequential logic besides combinational logic. In modern state-of-the-art implementations, logic cells are rather complex modules, including several LUTs, FFs, multiplexers, arithmetic and carry chains, and shift registers. In this way, complex logic functions can be implemented efficiently. In Xilinx's implementations, logic cells are referred to as configurable logic blocks (CLB) [1, 70], while in Intel's FPGAs they are referred to as logic array block (LAB) [32]. These logic cells constitute the main logic resources for implementing sequential and combinational logic functions.

Routing infrastructure and I/O blocks

In order to implement programmable computing fabric, logic cells are distributed in a regular 2D grid structure across the FPGA chip and connected trough a programmable routing infrastructure that allows communications between cells. The routing infrastructure is composed of switchable interconnects, which are typically implemented using multiplexers. In this way, logic cells can be connected together, forming a configurable fabric that can implement (given a sufficient amount of resources) any logic function. At the edges of the 2D programmable fabric, special input/output cells, allows connecting the logic cells to the outside world, providing the device with essential input and output capabilities.

Special-purpose cells

While a homogeneous fabric composed of logic cells could, in principle, implement any function, such an implementation would probably be rather

inefficient. In fact, many real-word algorithms require memory storage resources and often need to perform standard arithmetic operations. By implementing these kinds of resources as specialized hard cells, large classes of algorithms can be implemented more efficiently, minimizing the logic resources consumption. For this reason, the fabric of modern state-of-the-art FPGA is a heterogeneous structure in which arrays of logic cells are interleaved with arrays of arithmetic accelerator cells and dense on-chip random access memory cells. In Xilinx's UltraScale+ FPGA fabric, arithmetic accelerators cells are called digital signal processing (DSPs) blocks, while on-chip dense memory cells are called block RAM (BRAMs). Selected versions of the fabric may also include another type of memory cells, called UltraRAM, which allows even denser memory storage at the cost of a slower access rate. Depending on the specific version, the FPGA fabric may even include other types of special-purpose cells, like analog-to-digital converters, hi-speed transceivers, video encoders/decoders.

Figure 2.2 illustrates a schematization of the configurable logic fabric available on modern FPGA. Rows of logic blocks (LCs) are distributed across the FPGA chip interleaved by rows of specialized memory cells (MCs), and arithmetic/DSP cells (ACs). The cells are interconnected through a programmable routing infrastructure. Specialized I/O cells provide input/output capabilities.

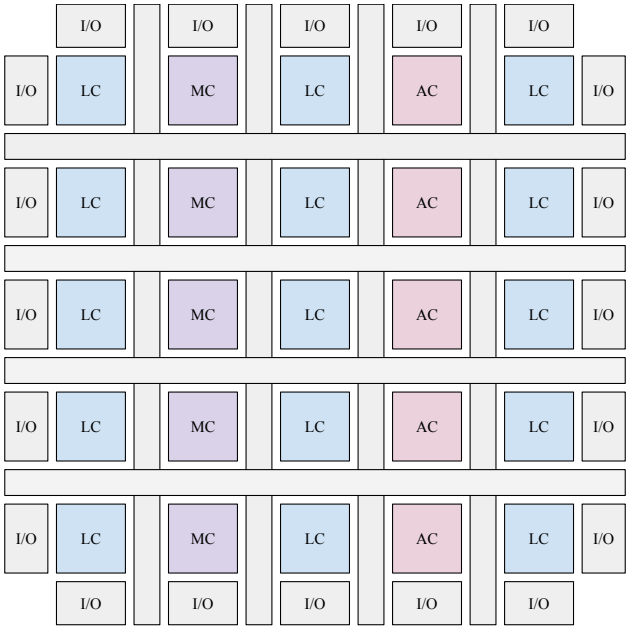

Figure 2.2: Schematization of the configurable logic fabric available on modern FPGA. Logic blocks (LCs) are distributed across the FPGA chip interleaved by rows of specialized memory cells (MCs), and arithmetic/DSP cells (ACs).

2.2 Design flow

A typical FPGA design flow can be summarized in two main steps (i) a design phase and (ii) a synthesis and implementation phase. The design phase consists of integrating a collection of logic entities into a system design. These entities could be first-party intellectual properties (IPs), created by the designer, or library IPs supplied directly by the FPGA vendor or provided by a third-party. In modern design flows, the designer can create its own IPs using a large variety of languages and tools, providing different levels of abstractions. On the one hand, hardware design experts can directly code IPs using low-level hardware description languages such as VHDL or Verilog, retaining full control over the generated hardware. On the other hand, software-oriented programmers can leverage HLS tools, which allows generating an RTL implementation staring form an algorithmic description coded using popular programming languages such as C or C++. Moreover, it is also possible to use model-based design tools, which allow creating IPs using automatic RTL code generation.

Once the design phase is complete, the resulting design comprising a collection of IPs in the form of HDL entities can be built through the synthesis and implementation phases. In the synthesis phase, the HDL design entities are translated into a set gate-level netlists describing the logic functions in terms of registers and logic gates. In addition to the gate-level netlists, the synthesis tool may also provide a netlist optimized for the specific target device describing the logic functions in terms of primitive logic cells available on the target architecture. Finally, the implementation phase maps the output products of the synthesis phase onto the physical resources of the FPGA. In the first step, the netlist gates are optimized and mapped to the primitive logic cells of the target FPGA fabric. Then, these primitives are placed on the physical FPGA fabric and routed, minimizing interconnections wires distances. Finally, the resulting placed and mapped design is translated into a binary file called bitstream.

2.3 FPGA configuration

Once the configuration bitstream has been generated, it can be transferred to the FPGA configuration memory through a configuration interface. Modern SoC-FPGA platforms typically include both internal and external configuration interfaces. External configuration interfaces allow configuring the FPGA from a host device using standard connections such as the JTAG interface. Conversely, internal configuration interfaces allow the FPGA to autonomously reconfigure itself without the need for a host machine. In SRAM-based FPGAs, the configuration memory is implemented using volatile memory cells that do not retain the state when the power is removed. For

this reason, bitstreams are often stored in a companion non-volatile memory, such as flash memory, connected with the FPGA. In this way, the FPGA can perform a self-configuration during the system boot process.

2.3.1 Dynamic partial reconfiguration

Dynamic partial reconfiguration is the ability to dynamically reconfigure a subset of logic cells included in the FPGA fabric while the remaining cells continue to operate without interruption [37, 73]. A physical subset of the FPGA fabric that can be dynamically reconfigured is referred to as a reconfigurable partition. The physical fabric of the FPGA can be divided into multiple reconfigurable partitions. Each partition is associated with a set of reconfigurable modules, which are design entities (such as IPs) synthesized and implemented for that partition. Each reconfigurable module typically performs a different functionality, and it is implemented into a configuration file called partial bitstream. A reconfigurable partition can host one of its reconfigurable modules at a time. The remainder of the FPGA fabric, which is not assigned to any reconfigurable partition, is referred to as the static region.

The amount of time required for reconfiguring a partition depends on the throughput of the FPGA's reconfiguration interface and the size of the partial bitstream. The size of a partial bitstream depends on the geometrical size of the reconfigurable partition and the kind of logic resources encompassed. Given a particular family of FPGAs, the throughput of the reconfiguration interface is usually bounded to a maximum rate depending on the specific fabric architecture. Hence, the reconfiguration time for a particular family of FPGA can be modeled as a function of the bitstream size. Usually, the reconfiguration time scales fairly linearly with the size of the partial bitstream and thus can be modeled using a linear function.

2.4 On-chip interconnections

As described in Section 2.2, a system design consists of a set of logic entities interconnected among them. These entities need to be connected among them to exchange data and control signals. In modern design flows, these entities are IP modules integrated and connect using an on-chip interconnect. The de-facto standard interface for on-chip interconnections in SoC-FPGA is the ARM Advanced Microcontroller Bus Architecture Advanced eXtensible Interface (AMBA AXI). The AXI standard defines a master-slave interface allowing simultaneous, bi-directional data exchange. The purpose of the AXI standard is to facilitate the development of complex FPGA designs with large numbers of components promoting the portability of IP cores. In the context of this thesis, AXI is the standard utilized for the development of the

framework proposed in Chapter 3, and the bandwidth reservation mechanism described in Chapter 5.

Chapter 3

The FRED framework

This chapter presents FRED, a framework for supporting predictable hardware acceleration of real-time applications using dynamic partial reconfiguration. FRED leverage dynamic partial reconfiguration for hosting in time-sharing a larger number of hardware accelerators than could otherwise be statically allocated on the physical fabric. The FRED framework is based on a platform model which is used to derive a response-time analysis for verifying the schedulability of a real-time task set under given constraints and assumptions. Although the proposed framework is based on a generic platform model, it has been conceived to account for several real-world constraints present on contemporary platforms. FRED has been prototyped on the Zynq SoC, showing that it can be supported by current SoC-FPGA platforms.

3.1 Platform model

FRED considers a 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 3.1. 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 $P = \{P_1, ..., P_{n_P}\}$, 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 between 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.

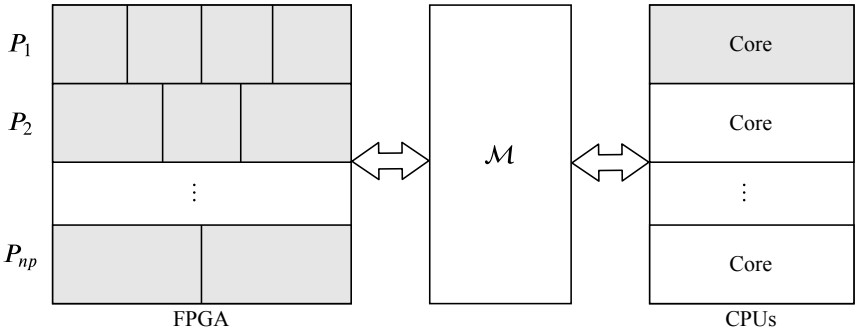

Figure 3.1: FRED platform model.

3.2 Application model

A FRED application consists of 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 by two sets of activities; (i) a set of n_S SW-tasks $\Gamma^S = \{\tau_1, ..., \tau_{n_S}\}$, and (ii) a set of n_H HW-tasks $\Gamma^H = \{\tau_1^H, ..., \tau_{n_H}^H\}$.

3.2.1 Hardware task model

Each HW-task $\tau_i^H \in \Gamma^H$ is an instance of an hardware accelerator requiring b_i 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 FPGA fabric's

slots.

3.2.2 Reconfiguration interface model

FRED assumes that the heterogeneous platform is equipped with an FPGA reconfiguration interface (FRI), which can dynamically reconfigure a slot at run-time forming a specific HW-task τ_i^H. Each slot can accommodate at most one HW-task [50, 9]. The FRI has been modeled to match the capabilities and limitations of real-world platforms (such as [37, 73]). 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. no processor cycles are used for reconfiguring a slot (the few clock cycles required for programming the FRI's DMA [77] are not modeled at this stage);
3. the FRI can reconfigure at most one slot at a time.

In order to reconfigure a given HW-task τ_i^H into a slot, the FRI has to reconfigure all its logic blocks, independently from the number b_i of logic blocks used 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 having affinity $P(\tau_i^H) = P_k$, it must be $b_i \leq b_k^S$.

The FRI is characterized by a *throughput* ρ, meaning that $r_k^S = b_k^S/\rho$ units of time are needed to 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 $r_a = r_k^S$.

3.2.3 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. In particular, each SW-task τ_i

- uses a set $\mathcal{H}(\tau_i) \subseteq \Gamma^H$ of m_i HW-tasks;
- alternates the execution of $m_i + 1$ sub-tasks (also referred to as chunks) with the execution of the m_i HW-tasks in $\mathcal{H}(\tau_i)$; thus, the execution of a SW-task τ_i can be represented as a sequence

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

  where $\{\tau_a^H, \tau_b^H, \ldots\} \in \mathcal{H}(\tau_i)$ and $\tau_{i,j}$ is the j-th sub-task of τ_i. Whenever the execution of a HW-task τ_a^H is requested, the corresponding SW-task *self-suspends* until the completion of τ_a^H. The beginning of the

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

Listing 3.1: Pseudo-code of the implementation skeleton of a SW-task.

self-suspension phase coincides with the termination of the sub-task that issued a request for a HW-task. In a dual manner, the completion of a HW-task coincides with the release of the next sub-task.

- has a total WCET C_i, composed of the WCETs $C_{i,j}$ of all its sub-tasks $\tau_{i,j}$; that is, $C_i = \sum_{j=1}^{m+1} C_{i,j}$.
- is periodically (or sporadically) released with a period (or minimum inter-arrival time)of T_i units of time, hence generating an infinite sequence of execution instances (denoted as jobs);
- is subject to timing constraints; that is, each of its jobs must complete its execution within a deadline D_i relative to its activation time.

Each HW-task can be used by at most one SW-task, that is $\bigcap_{\tau_i \in \Gamma^S} \mathcal{H}(\tau_i) = \emptyset$. Listing 3.1 reports the pseudo-code defining the body of a SW-task τ_i using $m_i = 2$ HW-tasks in the set $\mathcal{H}(\tau_i) = \{\tau_a^H, \tau_b^H\}$. The statement <...> represents a SW-task's generic code block containing a sequence of instructions that will be executed on the general-purpose processor.

The SW-task illustrated in Listing 3.1 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. `execute_hw_task(`τ_j^H`)` is a blocking system call, which is in charge of (i) requesting the execution of τ_j^H and (ii) suspending the execution of τ_i until the completion of τ_j^H. Note that at line 4, $\tau_{i,1}$ prepares the input data for τ_a^H. Similarly, $\tau_{i,2}$ retrieves the output data produced by τ_a^H (line 6) and prepares the input data for τ_b^H (line 8).

Figure 3.2 illustrates a timeline showing the delays experienced by another SW-task $\tau_i := \langle \tau_{i,1}, \tau_a^H, \tau_{i,2} \rangle$ when requesting the execution of a HW-task τ_a^H.

As visible in the figure, task τ_i is activated at time t_0. At time t_1, the first sub-task $\tau_{i,1}$ requests the execution of the HW-task τ_a^H and self-suspends its

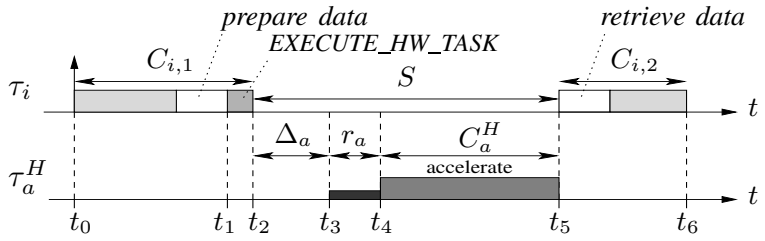

Figure 3.2: Execution behavior of a SW-task calling a HW-task. The *execute_hw_task* call issue the execution request to the scheduler.

execution at time t_2, where $(t_2 - t_1)$ corresponds to the system overhead to issue the request. This example assumes that all the slots of partition $P(\tau_a^H)$ are busy (i.e., currently occupied by other HW-tasks that are executing), hence a delay Δ_a is introduced from time t_2 until time t_3, at which one slot of $P(\tau_a^H)$ becomes free. Once there is a free slot in $P(\tau_a^H)$, the HW-task can be reconfigured, from time t_3 to t_4, using the FRI: such operation takes at most r_a units of time, where $r_a = b_k^S/\rho$ (being k the affinity of τ_a^H).

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

While the example presented above has a single SW-task, FRED considers applications consisting of multiple SW-tasks and HW-tasks that contend the resources available on the platform. Therefore, a SW-task τ_i can suffer a temporal interference from the execution of other SW-tasks that, if not properly managed, can determine the violation of its deadline D_i. Such interference also depends on the contention for the FPGA slots and the FRI caused by the other HW-tasks. For these reasons, a scheduling infrastructure is needed to support a set of concurrent HW-tasks and SW-tasks.

3.3 Scheduling infrastructure

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

Besides the processor, two other resources are contented by SW-tasks: the slots in the FPGA partition (shared with other HW-tasks having the

same affinity) and the FRI. Hence, multiple requests for such resources need to be scheduled. The overall scheduling infrastructure managing the slots and the FRI is based on a multi-level queue structure, illustrated in Figure 3.3.

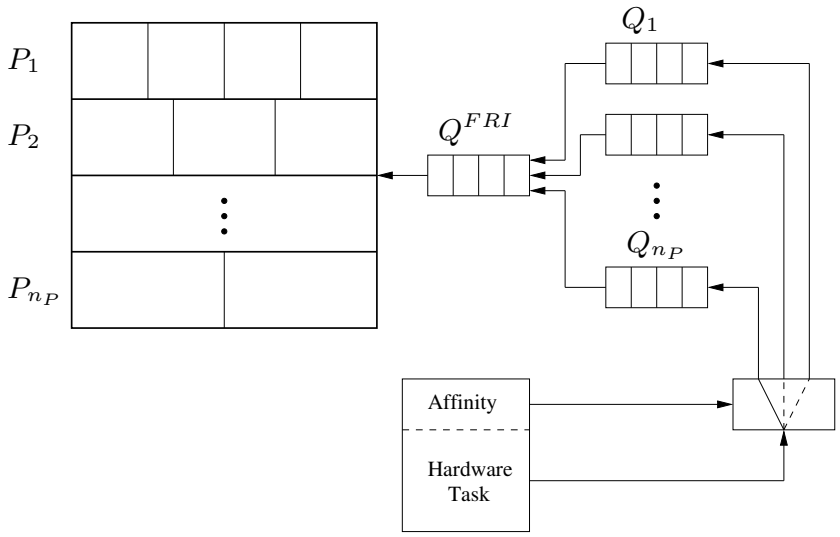

Figure 3.3: Scheduling infrastructure for HW-tasks requests in FRED.

The remainder of this section first describes the scheduling policies used for each resource. Then, it presents the scheduling rules that apply to every request for HW-tasks when traversing the multi-level queue structure of Figure 3.3.

3.3.1 Slot scheduling

For the purpose of scheduling, each slot can be *free* or *busy*. In turn, a busy slot can be *active* when there is a HW-task reconfigured on it that is executing or *reserved*. A HW-task τ_i^H with affinity $P(\tau_i^H) = P_k$, that is waiting for a free slot in partition P_k, is kept in a queue Q_k managed according to a first-in-first-out (FIFO) policy. Note that such a scheduling policy guarantees a starvation-free progress mechanism. Moreover, it does not require preempting the execution of HW-tasks, which is known to be a challenging issue [45][24] leading to non-negligible run-time overheads.

3.3.2 FRI scheduling

Whenever there are x free slots into a given partition P_k, such x slots are *reserved* for the first x HW-task requests waiting into Q_k, which then have to contend the FRI to reconfigure their corresponding HW-task. While slots are shared only among the HW-tasks belonging to the same partition, the FRI is a single resource contented by all the requests for HW-tasks in

the application. HW-task requests contending the FRI are kept in a queue denoted as Q^{FRI}.

In FRED, slot reconfiguration requests are managed according to a *ticket-based scheduling* policy, which is described below and can be configured to be executed either in a preemptive or non-preemptive mode. Please note that HW-task execution is assumed to be non-preemptive to contain the preemption overhead associated with FPGA configuration readback. Hence, preemptive and non-preemptive policies are only related to the FPGA reconfiguration phase through the FRI. When using a non-preemptive policy, the reconfiguration phase cannot be interrupted. On the contrary, when using a preemptive policy, the reconfiguration phase can be interrupted to serve another reconfiguration request.

3.3.3 Ticket-based scheduling

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

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

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

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

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

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

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

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

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

For non-preemptive FRI scheduling the following rules hold:

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

R-NP1 When the FRI completes a reconfiguration phase, or Q^{FRI} becomes not empty, the FRI starts reconfiguring the HW-task related to the first request in Q^{FRI}.

3.3.4 Scheduling example

Figure 3.4 shows an example of FRED preemptive FRI management schedule for an FPGA module containing two partitions P_1 and P_2, each consisting of a single slot.

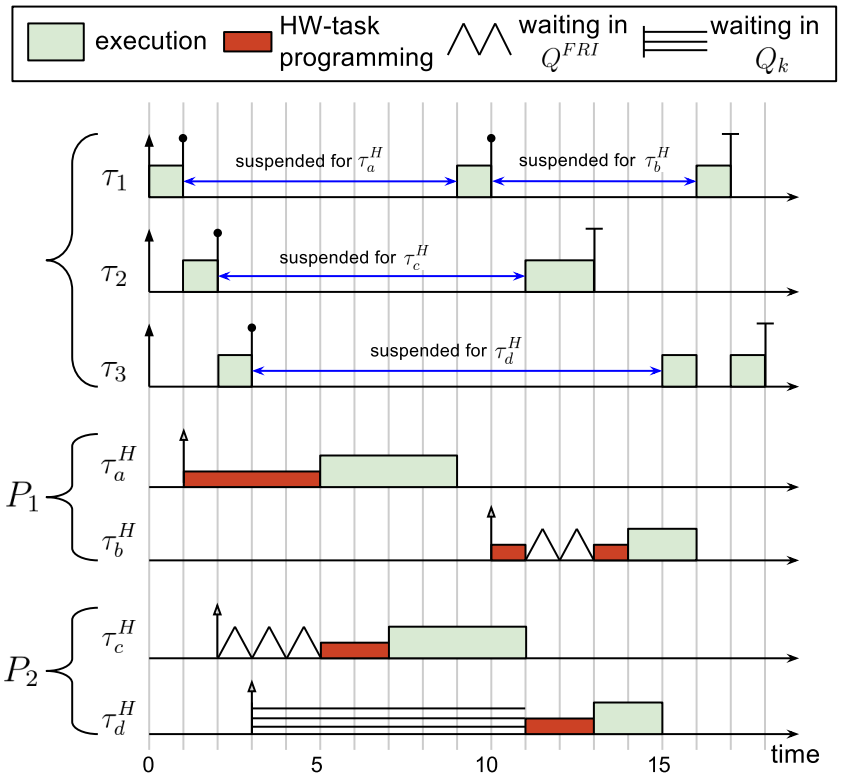

Figure 3.4: Example of FRED scheduling with preemptive FRI.

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

All the SW-tasks are synchronously released at time 0. Being the highest-priority one, τ_1 starts executing as first and at time $t = 1$ completes its

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

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

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

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

3.4 Communication between SW and HW-tasks

As stated in Section 3.2.3, SW-tasks make use of HW-tasks to accelerate specific computations; that is, a SW-task offloads a computation by requesting the execution of a HW-task on the FPGA and then retrieves the output of such a computation to continue the execution on the processor. As shown in Listing 3.1, the communication between a SW-task τ_i and a HW-task τ_a^H includes two phases:

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

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

It is worth observing that the approach used to enable communications between SW-tasks and HW-tasks can affect the real-time performance of the system by introducing different worst-case scenarios. For instance, suppose that the output data produced by a HW-task are stored in its internal memory area, and that phase (ii) comprises a copy from the local memory of the HW-task to a memory area accessible by the SW-task. In such a case, the HW-task must remain programmed onto the FPGA module until the sub-task in charge of executing the phase (ii) will be executed. Otherwise, output data would be lost.

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

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

(i) sub-task $\tau_{i,j}$ prepares the input data for τ_a^H in a memory area $\mathcal{M}_a^{\text{IN}}$ inside $\mathcal{M}$, and τ_a^H retrieves the input data by directly accessing $\mathcal{M}_a^{\text{IN}}$;

(ii) τ_a^H stores the output data into a memory area $\mathcal{M}_a^{\text{OUT}}$ inside $\mathcal{M}$, and $\tau_{i,j+1}$ retrieves them directly from $\mathcal{M}_a^{\text{OUT}}$, hence τ_a^H can release its slot as it finishes.

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

Please note that this communication strategy is not limited to platforms having a main memory shared between the processor and the FPGA, but it can also be used in platforms where a dedicated memory is reserved for such a communication. Indeed, the latter solution is more suitable for safety-critical systems requiring a higher level of predictability.

3.5 Response-time bounds

This section presents a summary of the FRED response-time analysis reminding to [12] for a comprehensive description since it is beyond the scope of this thesis. The FRED scheduling infrastructure presented in Section3.3 has been designed to ensure bounded response times by design. These bounds can be derived using a sufficient response-time analysis based on the response-time analysis for *real-time fixed-segment self-suspending tasks* (SS-tasks) developed by Nelissen et al.'s [49]. The SS-task model is a generic model for real-time computational activities where multiple execution phases are alternated to self-suspension phases. Hence, the SS-task model matches the execution behavior of the SW-tasks FRED precisely.

In a FRED application, each SW-task τ_i can be represented as a sequence $\tau_i := \langle \tau_{i,1}, \tau_a^H, \tau_{i,2}, \tau_b^H, \ldots, \tau_{i,m_i+1} \rangle$, where $\{\tau_a^H, \tau_b^H, \ldots\} \in \mathcal{H}(\tau_i)$ and $\tau_{i,j}$ is the j-th sub-task of τ_i. Each SW-task can be mapped to a segmented self-suspending sporadic task τ_s characterized by $m_i + 1$ computation segmens $C_{s,j}$ separated by m_i suspension intervals $S_{s,j}$ such as $\tau_s := \langle C_{s,1}, S_{s,1}, C_{s,2}, \ldots, S_{s,m}, C_{s,m+1} \rangle$ according to the following rules:

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

Intuitively, each sub-task of the SW-task τ_i is mapped into a sub-task of the SS-task τ_s, while each execution of a HW-task τ_a^H by τ_i is mapped to a suspension phase of the SS-task τ_s. Each HW-task suspension phase includes the reconfiguration time r_a, the WCET C_a^H, and the worst-case delay Δ_a suffered by the HW-task while traversing the FRED scheduling infrastructure. Finally, all other parameters of the SS-task τ_s are equal to the ones of the SW-task τ_i. By using these mapping rules, all SW-tasks parameters are known besides the delay Δ_a. Hence, to compute a safe upper-bound for SW-tasks response times, it suffices bounding the delay Δ_a.

In the case of preemptive FRI management, Δ_a can be bounded using the following theorem:

Theorem 1 (from [12])**.** *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. Using* preemptive *management of the FRI, the maximum delay Δ_a incurred by $\mathcal{R}_a$ is upper-bounded by*

$$\overline{\Delta_a^P} = \sum_{\tau_j \neq \tau_i} \max_{\tau_b^H \in \mathcal{H}(\tau_j)} \left\{ \Delta_b^{slot} + r_b \right\} \tag{3.1}$$

where

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

While, in the case of non-preemptive FRI management, a safe bound for Δ_a can be computed as follows

Theorem 2 (from [12]). *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*

$$\overline{\Delta_a^{NP}} = \overline{\Delta_a^{P}} + NH_k^{max} \times r_k^{max} \tag{3.2}$$

where

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

and

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

3.6 Practical validation and profiling

This section presents a preliminary prototype implemented on the Zynq-7000 platform to evaluate the feasibility of the proposed approach, profile hardware acceleration speedup factors, and measure reconfiguration overheads. The considered platform includes a dual-core ARM Cortex-A9 processor and a 7-series FPGA integrated on the same chip. The internal structure of a Zynq SoC can be divided into two main functional blocks referred to as processing system (PS) and programmable logic (PL) [77]. The PS block comprises the ARM Cortex-A9 MPCore, the memory interfaces, and the I/O peripherals, while the PL block includes the FPGA programmable fabric. The subsystems included in the PS are interconnected among themselves and to the PL through an ARM AMBA AXI (Advanced eXtensible Interface) interconnect.

The hardware modules configured on the PL can access the interconnect through a set of master and slave AXI interfaces exported by the PS side to the PL side. Slave interfaces allow modules to access the global memory space and share the DRAM memory with the processors. Dynamic partial reconfiguration is supported under the PS control. PL fabric can be fully or partially (re)configured by the PS through the device configuration interface (DevC) subsystem. The DevC includes a DMA engine that can be programmed to transfer bitstreams from the main memory to the PL configuration memory through the processor configuration access port (PCAP). Please note that since the DevC is included in the PS, it doesn't consume any PL logic resources to be instantiated.

3.6.1 System architecture

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

In general, since bitstream relocation is not supported by the Xilinx standard tools [37][73], each HW-task τ_i^H is implemented as a set of n_k^S bitstreams, one for each slot S_j of its associated partition $P(\tau_i^H)$. Each slot S_j can accommodate all the specific implementations of each HW-task τ_i^H that belongs to partition $P(\tau_i^H)$.

Since the slot interface should match the one of the HW-tasks [73], a common interface that all HW-tasks are required to implement has been defined. Such a common interface is similar to the one adopted by Sadri et al. [58]. The interface includes an AXI master interface for accessing the system memory, an AXI slave interface through which the HW-task can be controlled by the PS, and an interrupt signal to notify the PS. The AXI master interface logic allows HW-tasks to retrieve data autonomously from the memory space, implementing the communication mechanism described in Section 3.4.

In the current experimental setup, the AXI master interfaces exported by the HW-tasks are attached to high-performance slave ports exported by the PS, while the AXI slave control interfaces are attached to the general-purpose master ports. The software part consists of a user-level library for the FreeRTOS operating system. The library abstracts the reconfiguration mechanism and provides a simple API that enables SW-tasks to request the execution of HW-tasks on the PL through the `execute_hw_task()` function, as described in Listing 3.1.

3.6.2 Experimental setup

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

The developed case study includes four standard functions implemented both as HW-tasks and software code: three simple image convolution filters

(Sobel, Blur, and Sharp) and a matrix multiplier. The HW-tasks have been designed with Xilinx's Vivado HLS tool, while the software versions have been implemented in C99 language. The image processing HW-tasks process images of size 800×600 pixels, with 24-bit color depth. The matrix multiplier HW-task has been configured to multiply 512 elements integer matrices.

3.6.3 Experimental results

This section presents a set of experiments that have been performed to evaluate the feasibility of the proposed approach using the case study prototype.

Speedup evaluation experiment

A first experiment has been carried out to measure the speedup factors achievable from hardware acceleration on FPGA. The *longest observed execution times* (LOET) of the four HW-tasks have been measured and compared against the execution times of their software counterparts over 1000 runs. The results are reported in Table 3.1. The minimum speedup has been computed as the ratio between the minimum software execution time and the maximum hardware execution time observed. Despite the fact that the clock frequency (100 MHz) of the FPGA was slower than the one of the processor (650 MHz), hardware-accelerated implementations provided a relevant speedup between 5 and 15 over their software counterparts.

It is worth noticing that the measured speedup factors are dependent upon the specific implementation and optimization techniques. In general, it is reasonable to assume, for stream processing oriented operations, an average speedup factor ranging between 5 and 20, due to the high level of parallelism achievable on an FPGA.

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

Table 3.1: Speedup evaluation.

Response time experiment

A second experiment was carried out to evaluate the longest observed response times in a scenario where the number of HW-tasks exceeds the number of slots. The task set used for this test includes four SW-tasks that use the four HW-tasks defined in Section 3.6.2. SW-tasks are assigned

priorities according to the rate-monotonic algorithm. Table 3.2 summarizes the task parameters and the longest observed response times in an 8-hour run.

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

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

Table 3.2: Longest observed response times.

Figure 3.5 shows the distribution of the reconfiguration times observed in the previous scheduling test for more than 500,000 reconfiguration events. The longest observed reconfiguration time is 2.845 ms. Therefore, since all bitstreams have a size of 338 KByte, the minimum observed throughput of the PCAP configuration port resulted in being 116 MB/s, which is consistent with the throughput of 145 MB/s stated by Xilinx [77].

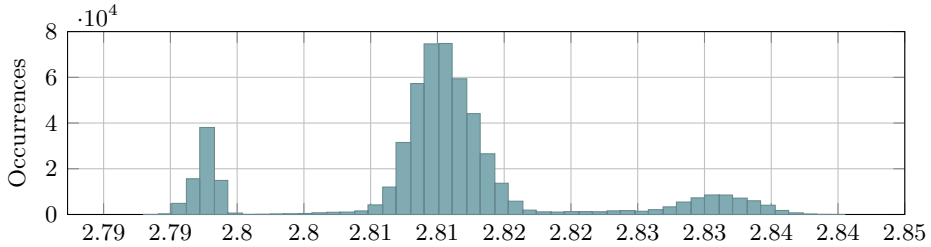

Figure 3.5: Distribution of reconfiguration times (ms).

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

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

Chapter 4

FRED on Linux

This chapter presents FREDLinux, an implementation of the FRED framework for Linux designed for Xilinx's heterogeneous multiprocessor system-on-chip (MPSoC) platforms. FREDLinux allows developing rich applications leveraging the large number of systems available on Linux (such as drivers, libraries, networking stacks, etc.) while relying on predictable FPGA-based hardware acceleration for performing heavy computations. FREDLinux is based on a runtime software support working on top of a system support design deployed on the FPGA fabric. The first part of this chapter presents the system design for implementing the FRED platform model. Then, the architecture of the software support for Linux is presented, comprising (i) a support for shared-memory communication with hardware accelerators, (ii) an improved driver to handle the FPGA reconfiguration, and (iii) a user-space server for managing hardware acceleration. Finally, this chapter presents a case study application designed for testing the proposed implementation in a realistic scenario.

4.1 Platform support

This section describes a design for supporting the FRED framework on over Xilinx's heterogeneous MPSoCs platforms, such as Zynq-7000 and Zynq UltraScale+, which have been chosen as the reference platforms for the framework. Xilinx's MPSoCs are popular heterogeneous platforms, including at least a multi-core ARM Cortex-A processor tightly coupled with a Xilinx's reconfigurable FPGA fabric. The internal structure of MPSoCs is divided into two main functional blocks: (i) the *processing system* (PS) block and, (ii) the *programmable logic* (PL) block [77]. The PS includes a multi-core Cortex-A processor, a set of memory controllers for interfacing external memories, a small amount of on-chip RAM, and various I/O peripherals. The PL side includes a reconfigurable FPGA fabric containing a different number and types of logic resources depending on the specific model. The subsystems included in the PS side, i.e., ARM cores, memory controller, and peripherals, are interconnected through an AMBA AXI bus. The same AXI infrastructure can be used to extend the system by connecting custom logic modules configured on the PL. The main interconnection between the PS and PL consists of a set of memory-mapped AXI interfaces exported by the PS side to the PL side.

4.1.1 System support design

The FRED support design provides the foundations for the software support enabling the deployment of dynamically-reconfigured HW-tasks on the PL fabric. Figure 4.1 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 hosting the hardware accelerators modules. The static region contains the AXI interconnection infrastructure, namely a set of AXI Interconnects (discussed in Section 4.1.1), and may host other support modules in an application-dependent fashion. The reconfigurable region is subpartitioned into a set of slots that are logically grouped in partitions following the specifications of the FRED framework presented in Chapter 3. As already discussed, a slotted approach is more suitable for real-time systems since no allocation and defragmentation overhead is introduced. Moreover, Xilinx's tools support static partitioning natively using Pblocks, which allows constraining implementation to a geometrical region of the FPGA. Hence a FRED design flow can be implemented using native design tools without relying on third-party experimental solutions for slotless support.

According to FRED's shared-memory communication paradigm, described in Section 3.4, each HW-task must be able to autonomously access memory regions that are also available to the processors. Xilinx's MPSoCs provides three alternatives for implementing such memory regions: (i) using the internal on-chip memory; (ii) using PL resources to build custom memories

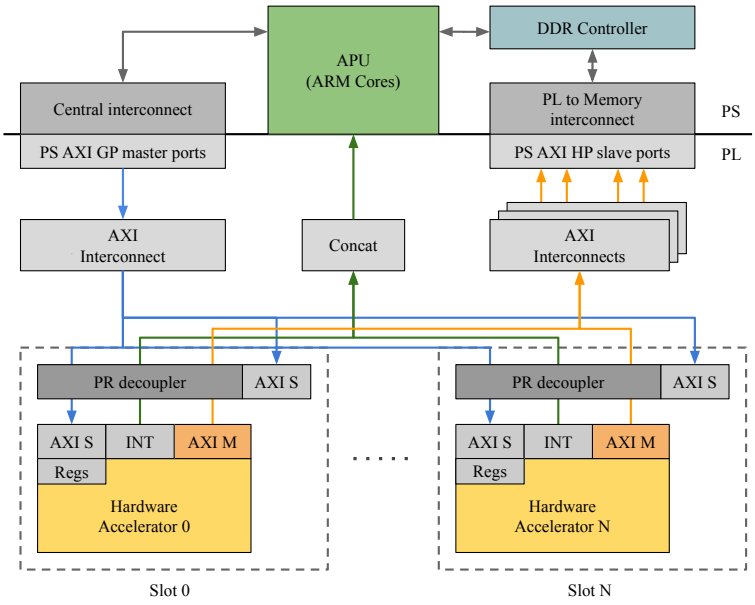

Figure 4.1: Support design for Xilinx's MPSoCs.

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 too small (256 KB) [77, 76] and hence may be unsuitable for supporting shared-memory communication with multiple HW-tasks. Alternative (ii) may determine a waste of the FPGA resources since implementing large memory buffers on PL require 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. The availability of such ports suggests that Xilinx's MPSoCs are well suited for designs that follow this approach. Hence, FREDLinux implements the shared-memory paradigm using the off-chip DRAM memory shared between the PS and the PL.

In FRED, each HW-task has affinity 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. This requirement can be fulfilled, within the constraints of real Xilinx's FPGA platforms [73], by defining a common interface that each HW-task must implement to be deployed on the system.

Common interface

The proposed common interface for HW-tasks consist 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 notify the PS cores. The AXI master interfaces (denoted as AXI M in Figure 4.1) allow HW-tasks to access the main memory through the PS DRAM controller, implementing the sharing memory paradigm. In this way, HW-tasks can autonomously retrieve the data they need to process without hampering the processor. Besides the FRED requirements, *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 4.1) allows mapping HW-task's control and data register into the ARM cores address space. In this way, HW-tasks can be controlled by the FREDLinux software support. All HW-tasks are required to implement the same set of control and data registers map, allowing the software support to use a common software driver. The eight data registers are used to exchange memory pointers and their meaning depends on the specific function implemented by the HW-task. Finally, the interrupt signal (denoted as INT in Figure 4.1), is meant to be connected to the interrupt controller to notify the completion of the HW-task to the ARM cores.

FREDLinux has been designed with high-level synthesis support in mind to enable the implementation of computationally intensive functions as FPGA-based hardware accelerators. Hence, compliant HW-tasks can be easily generated using the popular Vivado HLS tool, starting from a high-level behavioral description. To this end, it is sufficient to wrap a C or C++ function into the common top-level wrapper reported in Listing 4.1 for generating a FREDLinux compliant HW-task. The HLS tool will automatically generate the standard interface logic thanks to its interface synthesis capabilities. In addition to HLS, it is also possible to code HW-tasks directly using hardware description languages such as VHDL or Verilog for achieving higher performances. A VHDL stub is provided in such a case. Moreover, Xilinx's Vivado suite provides HDL code stubs for implementing the AXI master and AXI-Lite slave interfaces.

Dynamic partial reconfiguration support

In Xilinx's MPSoCs, the FPGA fabric within the PL can be fully or partially (re)configured under the control of the software running in the ARM cores inside the PS 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 main DRAM memory to the PL configuration memory. Compared to other configuration paths, such as the *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. Xilinx's standard toolchain does not support bitstreams relocation [73], i.e., the same bitstream cannot be used to program the same HW-tasks in different slots. This limitation can be overcome by synthesizing

Listing 4.1: HLS code for implementing HW-tasks.

```
void slot_i(args_t *id, args_t args[ARGS_SIZE], volatile data_t *mem_in,
    volatile data_t *mem_out)
{
  // AXI Lite control bus
  #pragma HLS INTERFACE s_axilite port=return bundle=ctrl_bus
  #pragma HLS INTERFACE s_axilite port=id bundle=ctrl_bus
  #pragma HLS INTERFACE s_axilite port=args bundle=ctrl_bus

  // AXI Master memory ports
  #pragma HLS INTERFACE m_axi port=mem_in offset=slave bundle=mem_bus
  #pragma HLS INTERFACE s_axilite port=mem_in bundle=ctrl_bus
  #pragma HLS INTERFACE m_axi port=mem_out offset=slave bundle=mem_bus
  #pragma HLS INTERFACE s_axilite port=mem_out bundle=ctrl_bus

  fred_hwacc_body(id, args, mem_in, mem_out);
}
```

a different bitstream for each slot of the partition to which the corresponding HW-task belongs. It is worth noting that this approach does not imply a large memory consumption, as bitstreams files are typically in the order of a few megabytes.

Slot decouplers

During the FPGA reconfiguration process, the behavior of the reconfigurable slot is undefined since its logic cells may be in an inconsistent state. Hence the logic cells may generate temporary glitches causing cause troublesome spurious transactions in other modules such as the AXI interconnects, or the ARM cores interrupt controller. To solve this problem, each slot is protected by a partial reconfiguration decoupler (denoted as PR decoupler in Figure 4.1), which binds slot's interface wires to safe logic values during the reconfiguration process [73]. The FREDLinux runtime controls each decoupler through a single control register, which is mapped into the address space through an AXI-Lite slave interface.

Interconnections

In FRED, all HW-tasks employs bus mastering techniques for accessing the system memory and sharing data with SW-tasks running on the ARM cores. Hence, bus and memory access represents a crucial contention point. The problem of controlling bus and memory contention in a predictable fashion is addressed later in this thesis since it requires the development of custom hardware modules for custom managing the AXI bus. In FREDLinux, the rationale behind the interconnection strategy employed for the system

support design is to evenly distribute the bandwidth supplied by all available HP ports to the HW-tasks. To this end, the AXI master interfaces exported by each slot (using a placeholder HW-task) are connected to an interconnect block [8] associated with the slot, thus resulting in a single AXI master interface. If the number of HP ports actually available from the PS is less than the number of slots, the slots' interconnects master interfaces are connected to the HP ports directly. Otherwise, if the design contains more HW-task than available HP ports, an additional level of interconnects is required to connect the slots' interconnects master interfaces to the available HP ports. Please note that the support design depends only on the total number of slots and not on the actual number of HW-tasks.

4.2 Linux support

This section describes the architecture of the FREDLinux software support built on top of the system support design. The software support has been designed in a modular fashion, relying as much as possible on user-space implementation for improving maintainability, safety, and extendability. The internal architecture of the system is shown in Figure 4.2. The central component of the software support is a user-space server process, named fred-server, who is in charge of handling and dispatching acceleration requests from SW-tasks. The fred-server relies upon two custom kernel modules, and the UIO framework, for performing the low-level operations required to control the hardware components of the system support design.

Periodic SW-tasks can be implemented as regular Linux processes or threads using the POSIX compliant SW-task body presented in Listing 4.2. The SW-task body iteratively (i) performs its computations calling one or more HW-tasks, and then (ii) make 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. The need for implementing the FRED shared buffers paradigm, described in Section 3.4, requires the development of an efficient mechanism to share data between the virtual and the physical domains. Recent MPSoCs 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 does 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, FREDLinux relies on a zero-copy design, using coherent memory buffers as communication channels between SW-tasks and HW-tasks. The zero-copy design allows SW-tasks and

HW-tasks to share data without any associated overhead.

Listing 4.2: Pseudo-code stub for a SW-task.

```
void sw_task_stub(void *args)
{
  struct timespec ts;
  int period_ms = <task_period>;

  /* Get current time */
  clock_gettime(CLOCK_MONOTONIC, &ts);
  /* Set next activation */
  time_add_ms(&ts, period_ms);

  while (1) {
    /*
     * SW-task body:
     * <First software chunk>
     * <Call HW-task>
     * <Second software chunk>
     * <Call HW-task>
     * <Third software chunk>
    */

    /* Sleep until next activation */
    clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &t, NULL);
    /* Set next activation */
    time_add_ms(&t, period_ms);
  }
}
```

4.2.1 Kernel-space components

The fred-server process uses the two custom kernel modules for (i) allocating the memory buffers employed to share data between SW and HW-tasks, and (ii) managing the PL fabric reconfiguration. The fred-server relies on the UIO framework for managing HW-tasks, i.e., accessing control and data registers, and observing the interrupt lines.

Buffers allocator module

To enforce memory coherence between SW-tasks and HW-tasks, the *shared-memory* infrastructure described in Section 3.4, has been implemented using a set of uncached memory buffers allocated by a custom kernel allocator module. Such a module uses the Linux DMA layer to allocate physically contiguous (uncached) memory buffers used 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 the fred-server during the initialization phase for requesting the allocation of memory buffers. Each allocation request is performed using the `ioctl` syscall, including, as a parameter, 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 user-space.

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 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. Please note that there are no cache levels common to both the processor and the hardware accelerators when directly connected to the DRAM controller trough the HP ports. On the other side, a HW-task can access the same buffer through a physical memory address. The buffers' physical address are passed to the HW-tasks by the FRED server using the set of data registers belonging to the HW-tasks' common interface described in Section 4.1.1.

In this way, data can be efficiently shared between HW and SW tasks without any copy operation or operating system overhead. It is worth observing that, with this design, the SW-tasks never directly deal with memory management operations. SW-tasks see their shared buffers only as a set of character device that can be mapped, during its initialization phase, into their virtual memory spaces. From the programmer's perspective, the process of mapping of these buffers, likewise all other interactions with the fred-server, is assisted by the client support library described in Section 4.3. During the system shutdown phase, the fred-server releases the buffer devices created during the initialization phase using the `ioctl` syscall again on the `fred_buffctl` control device.

Reconfiguration module

This section presents an optimized reconfiguration driver based on Xilinx's original kernel module for enabling dynamic partial reconfiguration under Linux. Such a kernel module allocates a character device named `xdevcfg` that can be used to reconfigure at run-time the FPGA fabric from user-space using a bitstream as input. Once loaded, the kernel module instantiates

a character device named `xdevcfg`. The reconfiguration process can be initiated by writing the bitstream file into the `xdevcfg` device file using a standard `write()` operation. The Xilinx's original kernel module 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 an 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 is completed. This design is clearly intended to keep the driver safe and easy to use. Still, it is clearly unsuitable for FRED since the overhead caused by multiple user-to-kernel copies and busy waits is not compatible with the intensive usage of partial reconfiguration required by the framework. To overcoming these issues, the original driver has been modified, taking advantage of the allocator module described in the previous section. The rationale is to pre-load all the HW-tasks' bitstreams 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 into the physically contiguous memory buffers, they can be reached by the internal DMA engine used for partial reconfiguration. For this reason, the original kernel module has been modified by adding an `ioctl()` method that allows starting the reconfiguration by passing a memory reference to a pre-allocated bitstream to the driver. In order to avoid the busy-wait, the `poll()` method of the character device interface implemented, providing support for I/O multiplexing. 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 these modifications, the reconfiguration process is started by an `ioctl()` call on the `xdevcfg` device, which returns immediately. Then, a user-space application, like the fred-server, can be notified for the conclusion of the reconfiguration without busy-waiting through I/O multiplexing.

4.2.2 User-space components

The fred-server is the central user-space component of FREDLinux. Form an architectural perspective, the fred-server is an event-driven application that handles service requests coming from multiple event sources like SW-tasks performing acceleration requests, HW-tasks notifying their completion, and

other hardware events like the conclusion of the FPGA reconfiguration process. From a functional perspective, 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 to communicate with Linux and the kernel support, as illustrated in Figure 4.2. The communication interface between the fred-server and SW-tasks is implemented using UNIX domain sockets. In this way, SW-tasks are decoupled from the fred-server.

During the initialization phase, the fred-server reads a set of files describing the system layout and the available HW-tasks. Then, according to such a system description, 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 the fred-server are encapsulated into the client support library to ease the development process. It is worth noticing that SW-tasks never interact directly with the hardware, nor are they required to perform privileged operations. The fred-server mediates any interaction between client SW-tasks and the platform hardware.

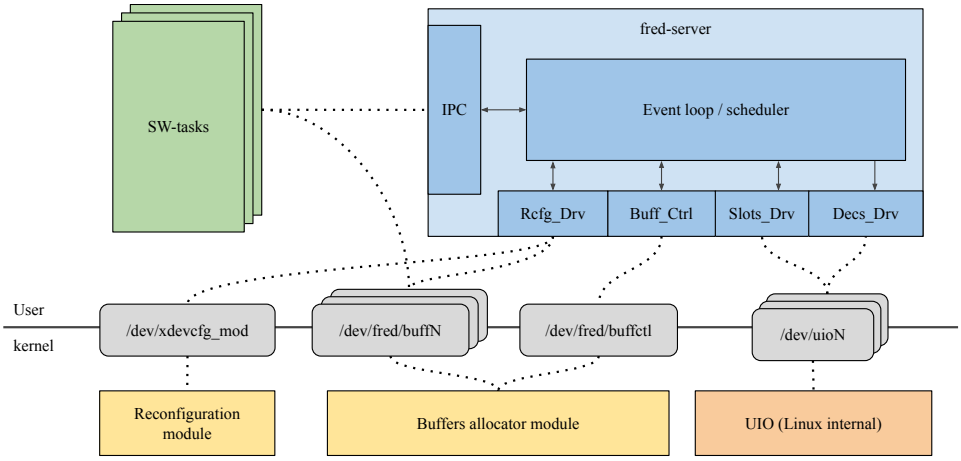

Figure 4.2: Overview of the fred-server.

Fred-server internals

The fred-server is written in standard C99 using standard POSIX and Linux API. In recent versions, 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 arriving concurrently from

multiple event sources [62, 54]. In particular, the rector pattern decouples the responsibility of receiving and demultiplexing events from the responsibility of actually handling events. This characteristic is especially useful within FRED since multiple software and hardware event sources like SW-tasks, HW-tasks, FPGA reconfiguration interface, etc. need to be handled differently. 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 [43] 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 [43] 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 [60, 63]. The internal architecture of the fred-server is illustrated in Figure 4.3, highlighting the most relevant interactions between its key internal components.

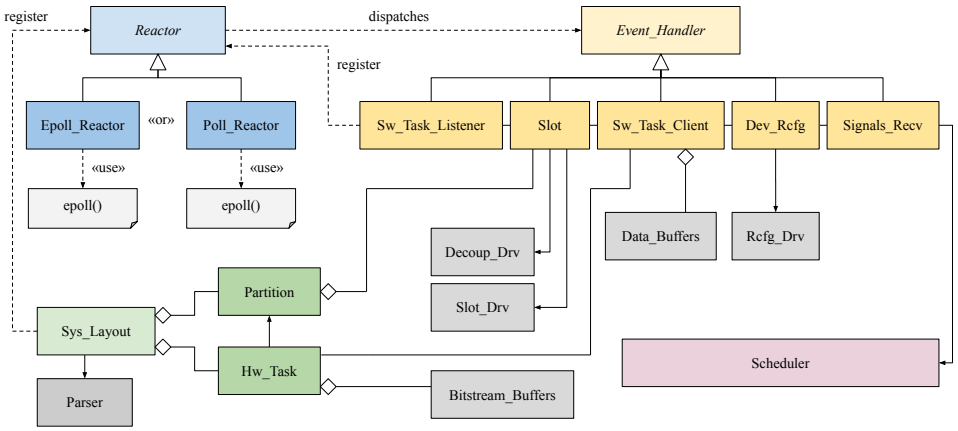

Figure 4.3: Internal components of the fred-server.

- 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 holds the responsibility of serving a specific type of event. All event handler instance contains 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 implementation 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, FREDLinux 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 POSIX's standard `poll` function. 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 `poll` function scales up linearly (i.e., $\mathcal{O}(n)$) with the number of handles. However, given the limited number of event sources present in a typical FREDLinux design, and considering the limited amount of overhead introduced by the whole event handling logic, the performance difference is very limited in practical cases.

- The `Software_Task_Listener` is a concrete event handler which 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 and registers to the reactor a new `SW_Task` object that handles the new connection socket.

- The `Sw_Task_Client` 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` class is a concrete handler representing a "physical socket" for HW-task. That is 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, the reconfiguration process is skipped to save time. The `Slot`

object controls the contained HW-task and the slot's paired decoupler using, in turn, the `Slot_Drv` and `Decoup_Drv` low-level components. These components allow decoupling the FRED control logic from the responsibility of performing the low-level control actions, which depends 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 then the contained HW-task completed its execution.

- The `Dev_Rcfg` 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` is an optional concrete handler that can be used for handling inter-process communication signals synchronously with the reactors' event loop logic 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 `Scheduler` module is the central component in charge of implementing the FRED scheduling policy described in Chapter 3 using the proposed multi-level queue structure. All event handlers components notify the scheduler when an event occurs. In turn, the scheduler performs the required actions.

- The `Sys_Layout` class models the physical layout of the system. A FREDLinux system is composed by 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 configurations files describing the FREDLinux 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 instantiating all 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`.

4.3 Client support library API

The client support library provides a lightweight API that can be used by client programmers to develop FREDLinux hardware-accelerated applications. The client support library presented here is written in C99, and it is designed to provide support for C or C++ applications. However, since SW-tasks are completely decoupled from the fred-server through UNIX domain sockets,

additional client API for other languages can be easily developed as long as they follow the standard communication protocol with the fred-server. For instance, a Python implementation is already available, although it is not presented here since it's beyond the scope of this thesis.

Listing 4.3: 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 **hw_task,
       uint32_t hw_task_id);
9
10 int fred_accel(struct fred_data *self, const struct fred_hw_task *hw_task);
11
12 void fred_free(struct fred_data *self);
13
14 //------------------------------------------------------------------------
15
16 int fred_get_buffs_count(const struct fred_data *self, struct fred_hw_task *
       hw_task);
17
18 ssize_t fred_get_buff_size(const struct fred_data *self, struct fred_hw_task
        *hw_task, int buff_idx);
19
20 //------------------------------------------------------------------------
21
22 void *fred_map_buff(const struct fred_data *self, struct fred_hw_task *
       hw_task, int buff_idx);
23
24 void fred_unmap_buff(const struct fred_data *self, struct fred_hw_task *
       hw_task, int buff_idx);
```

Listing 4.3 reports the functions composing the client support library API. The `fred_init` function initiate the communication with the fred-server initializing an opaque handler of type `struct fred_data` which 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 an initialize 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 process' address space 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 respectively the number and the size of the buffers used by an HW-task. Once the SW-task has completed its initialization phase, binding with the HW-tasks and mapping the associated data buffers, it can proceed with is computations, eventually filling the shared buffers and calling one or more HW-task using the `fred_accel` function. The `fred_accel` is a blocking function that suspends the SW-tasks until the invoked HW-task completes its execution. 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. 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 the fred-server by calling the `fred_free` function. Listing 4.4 shows the pseud-code of a SW-task implemented using the C API provided by the client support library. For the sake of clarity, the SW-task uses only a single HW-task, and errors handling code is omitted.

Listing 4.4: Pseudo-code stub of a SW-task using the C API.

```
void sw_task(void *args)
{
  struct timespec ts;
  int period_ms = <task_period>;

  struct fred_data *fred;
  struct fred_hw_task *hw_task;
  uint32_t hw_task_id = <hw_task_id>;

  void *buff_in = NULL;
  void *buff_out = NULL;

  /* Initialize communication and bind a HW-task */
  fred_init(&fred_data);
  fred_bind(fred_data, &hw_task, hw_task_id);

  /* Map the buffers */
  buff_in = fred_map_buff(fred, hw_task, 0);
  buff_out = fred_map_buff(fred, hw_task, 1);

  /* Get current time */
  clock_gettime(CLOCK_MONOTONIC, &ts);
  /* Set next activation */
  time_add_ms(&ts, period_ms);

  while (1) {
    /* Fill input buffer */
    buff_in[i] = <....>

    /* Call the HW-task */
    fred_accel(fred_data, hw_task);

    /* Read output data buffer */
    <....> = buff_out[i]

    /* Sleep until next activation */
    clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &t, NULL);
    /* Set next activation */
    time_add_ms(&t, period_ms);
  }
}
```

4.4 Case study application

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 multiplication. Besides FREDLinux, the case study application make use of Video4Linux (V4L) and Qt frameworks available on GNU/Linux. The case study has been developed and tested on 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 custom hardware accelerators (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 regular OpenCV C++ API. The filters are implemented as a stack of OpenCV functions inspired by Xilinx's recommendations described in [3]. Figure 4.4 shows the output of the image filters using a test image as input. Finally, the last processing function is an integer matrix multiplier implemented both in HLS and C++ using the naive $\mathcal{O}(n^3)$ algorithm.

The amount of logic resources required to allocate all processing functions HW-tasks statically (i.e., allocated at the same time) exceeds the amount of resources available on the physical FPGA. Hence, using hardware acceleration for all tasks would be unfeasible. FREDLinux allows to "virtually" extend the amount of logic of resources enabling the platform to accommodate all the accelerators in timesharing.

4.4.1 Case study architecture

From a system perspective, the application is composed of 4 SW-task and 4 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 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 calling the corresponding HW-task to perform the computation.

In the default configuration, the image filter processing functions are set for processing images of 640×480 pixels with 24-bit color depth, while the matrix multiplier performs 30 multiplication of 64×64 matrices. The hardware implementations of processing functions have been wrapped within the FREDLinux standard HW-task interface presented in Listing 4.1 and translated into RTL implementations using Vivado HLS. At the same time, the equivalent software versions are compiled as regular C++ functors or

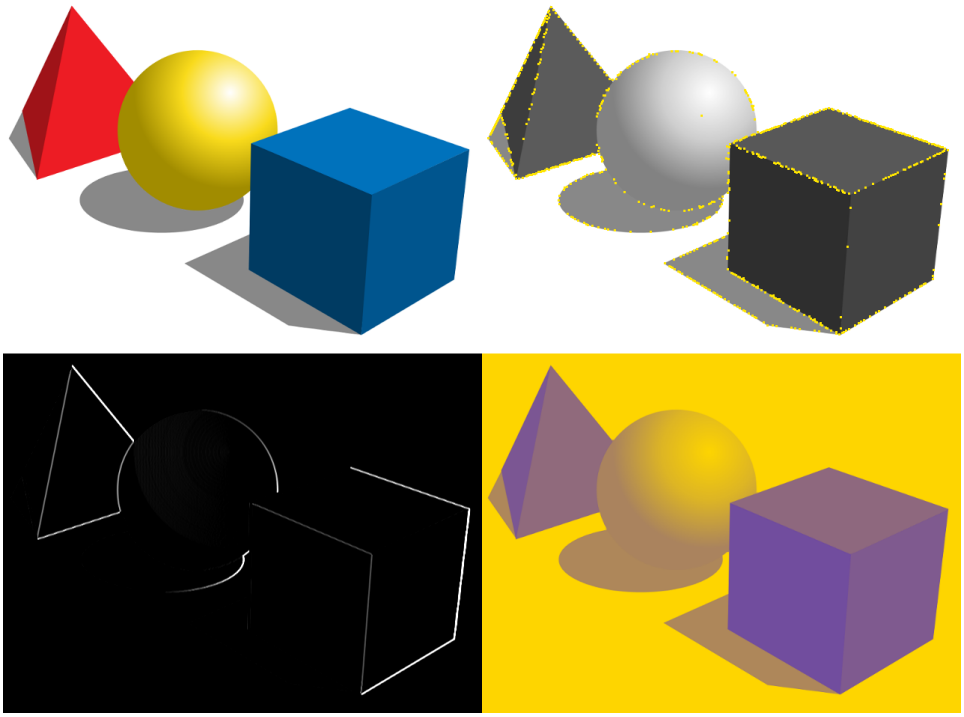

Figure 4.4: Test image processed using filter processing functions. Clockwise from the top: Test image, FastX filter, gradient map (Gmap) filter, Sobel filter.

functions.

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 4.5. The frame grabber thread copies the frames acquired by the webcam into a shared buffer implemented as a *cyclic asynchronous buffer* (CAB) [15]. 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 OpenCL 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 the fred-server, and the SW-task is suspended. After the completion of the HW-task, the owner SW-task resumes its execution and retrieves the

processed frame form the HW-task's output buffer, which is associated with a `QImage` object to avoid another additional copy. Finally, independently from hardware or software mode, the resulting image is stored in a `QImage` buffer that can be passed to the Qt window component to be displayed.

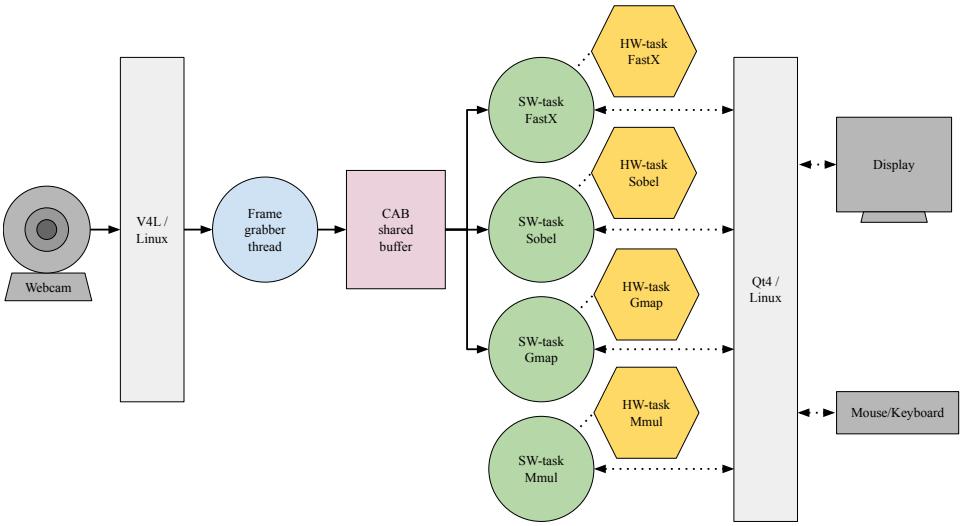

Figure 4.5: Overview of the application structure.

4.4.2 Application internals

The case study application has been implemented in C++11 using Qt and V4L on top of the Xilinx's Petalinux "distribution". Figure 4.6 presents an overview of the application's internal architecture. All SW-tasks are subclasses of the abstract base class `AbsTask` that realize a functor (callable object), with a periodic behavior, dispatchable to a standard `std::thread` object. The image processing SW-tasks are instances of the derived class `MultiFrameProcessor` that owns a set of `AbsFrameFilter` objects. These objects are filter components used for processing sequences of images and include the logic for interfacing with the Qt framework. Among the set filter objects components, only one can be active at a time. The active filter object can be exchanged at runtime to change the behavior of the frame processor.

The `AbsFrameFilter` abstract base class is subclassed into the `SwFrameFilter` and `HwFrameFilter` concrete classes providing the logic for implementing, respectively, a pure software and a hardware-accelerated filter. Once instantiated, these classes are associated are the actual filtering functor object, in the case of a software filter, or a `HwTask` object in the case of a hardware filter. The `HwTask` object is a functor object that wraps all the logic required to interact with the fred-server using the client support library presented in Section 4.3. Each `HwTask` instance is bounded to a HW-task by passing the

corresponding HW-task's id to the class's constructor.

In the current version of the application, only two frame filters are attached to each frame processor SW-task; (i) an OpenCV software implementation of the processing function filter and (ii) a hardware implementation of the same filter in the form of a HW-task. The matrix multiplier SW-task is an instance of the `MultiFuncTask` class, which is dedicated to nonvisual processing and contains a set of processing functor as components. Similarly to the image processing tasks, only two functors are attached in the current version (i) a software implementation in the form of a regular function, and (ii) a hardware implementation of a matrix multiplier int the form of a HW-task.

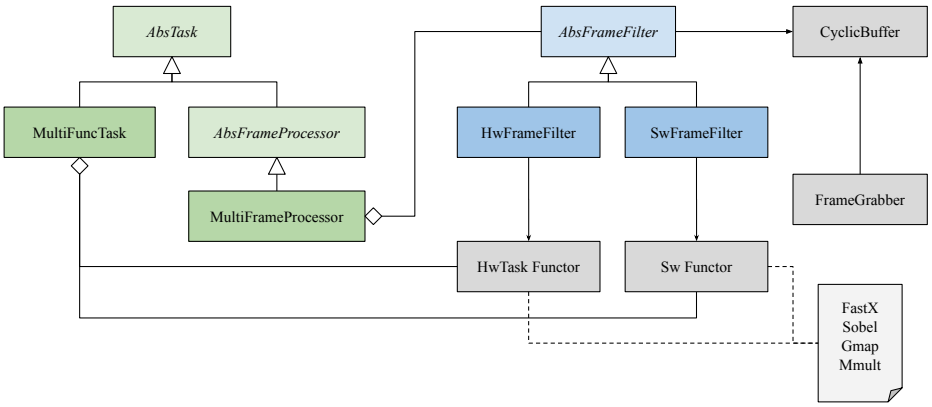

Figure 4.6: Informal class diagram of the application.

Programmable logic partitioning

The Zynq PL FPGA fabric is divided into a static region and a reconfigurable region according to the FREDLinux support design described in Section 4.1.1. The static region contains a set of AXI Interconnect 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 accounting for 5600 LUTs, while the second partition P_1 includes 14% of the total logic resources corresponding to 2400 LUTs. The remaining resources, 9600 LUTs, corresponding to approximately 54% of the total, are reserved for the static region. A representation of the partitioning is presented in Figure 4.7. The total resource distribution is slightly more uneven since special-purpose cells like DSPs and BRAMs are not homogeneously distributed on the fabric.

The FastX and the matrix multiplier HW-tasks are associated with the larger partition P_0 requiring more resources. The lighter Sobel and Gmap

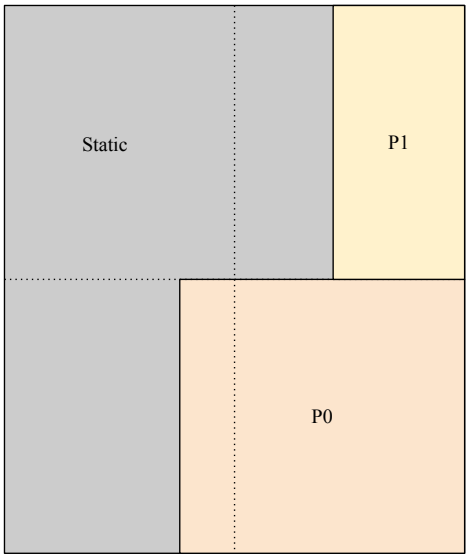

Figure 4.7: Visualization of the FPGA fabric partitioning.

HW-tasks are associated with the smaller partition P_1. Since both partitions are composed of a single slot, only one bitstream is required for each HW-task.

4.5 Performance evaluation

This section presents the results of a set of experiments aimed at evaluating the performances of FREDLinux using the case study application.

4.5.1 Speed up evaluation experiments

This first set of experiments has been carried out to evaluate the speedup achieved for each processing function introduced through hardware acceleration in comparison to a pure software implementation. To this end, for each processing function, a separate experiment has been put in place. In each experiment, the application has been configured to run only a single processing SW-task to avoid interferences. Each trial consists of two separate runs. In the first run, the SW-task is configured to run in hardware mode, which means that the SW-task executes the processing function by calling the corresponding HW-task computing on the FPGA. In the second run, the SW-task is configured to run in software mode, which means to call the software implementation of the processing function running on the ARM Cortex processor. Each experiment runs consist of more than 10000 jobs.

During both runs, 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, the fred-server, and Linux is still present. The results of the experiments are summarized in Table 4.1. It 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 considering that all HW-task filters process images of the same size, i.e., 640×480 pixels with 24-bit color depth. Therefore, they move the same amount of data from and to the DRAM system memory.

Function name	SW avg. [ms]	SW worst [ms]	HW avg. [ms]	HW worst [ms]	Avg. speed up
FastX	58.222	62.864	4.905	5.068	11.869
Gmap	55.521	56.690	4.770	4.879	11.639
Sobel	68.823	71.119	4.864	4.976	14.146
Mmul	65.080	70.112	23.662	23.748	2.751

Table 4.1: Speed up evaluation experiments.

4.5.2 System acceleration experiment

A second experiment has been then carried out to evaluate the performance improvement achievable with hardware acceleration considering the whole application. For this purpose, the application is configured to run with all four SW-tasks described in Section 4.4.1 active at the same time. In order to evaluate the system speedup provided by the FREDLinux hardware acceleration, 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 the 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 blocking because there are no free slots currently available or the reconfiguration interface is busy. The parameters of all software activities involved in this test, including SW-tasks, are summarized in Table 4.2. The results of the experiment are reported in Table 4.3 comparing the observed response times of the SW-tasks while running in software and hardware modes for over 30 minutes. Figure 4.8 presents a distribution of the response times.

Overall, the results of this practical evaluation show that the resource "virtualization" mechanism provided by FREDLinux allows improving the performance of case study application thought hardware acceleration even when it would be unfeasible with a static approach due to logic resources limitation.

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 4.2: Software activities parameters.

Activity name	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 4.3: Comparison of the SW-tasks's observed response times in software and hardware modes.

Overhead evaluation

The second object of this experiment is to evaluate the system overhead introduced by the fred-server using the proposed case study application. The actual overhead introduced by the fred-server vary depending on the specific version and may change as the development process continues. Hence, this experiment aims at estimating the overall impact of the fred-server on the system compared to other activities. The overhead experienced by a SW-task due to the fred-server is visualized in figure 4.9. The first component $O_{j,1}$ includes (i) the communication overhead introduced by handling a new request, (ii) the overhead experienced by the request while traversing the multi-level queue structure and, (iii) the overhead introduced by programming the reconfiguration interface. The second component $O_{j,2}$ includes the overhead introduced for reactivating the fred-server and programming the control and data registers of the hardware accelerator module. Finally, the third component $O_{j,3}$ accounts for the overhead introduced by the reactivation of the fred-server and the communication for notifying the completion of the acceleration request to the SW-task.

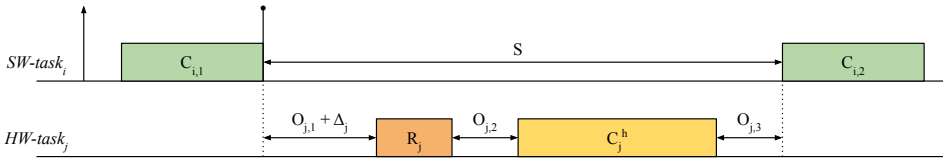

Figure 4.9: Execution behavior of a SW-task calling a HW-task considering the overheads introduced by the FRED server.

It is worth noting that, when considering the whole application consisting of multiple SW and HW-tasks, those overhead components depend on the number of concurrent requests that the server must handle. This because the fred-server is activated every time an event source (.e.g., SW-task acceleration requests, HW-tasks execution complete, FPGA reconfiguration completes) becomes active. While serving these events, the fred-server makes acceleration requests progressing through scheduling multi-queue infrastructure up to the reconfiguration and execution stages. Hence, the overhead components experienced by a SW-task experience are inflated by the cost of managing other acceleration requests. The results of this experimental evaluation show that the fred-server process is running for the 0.945% of the total time in the entire 30 minutes hardware-accelerated run.

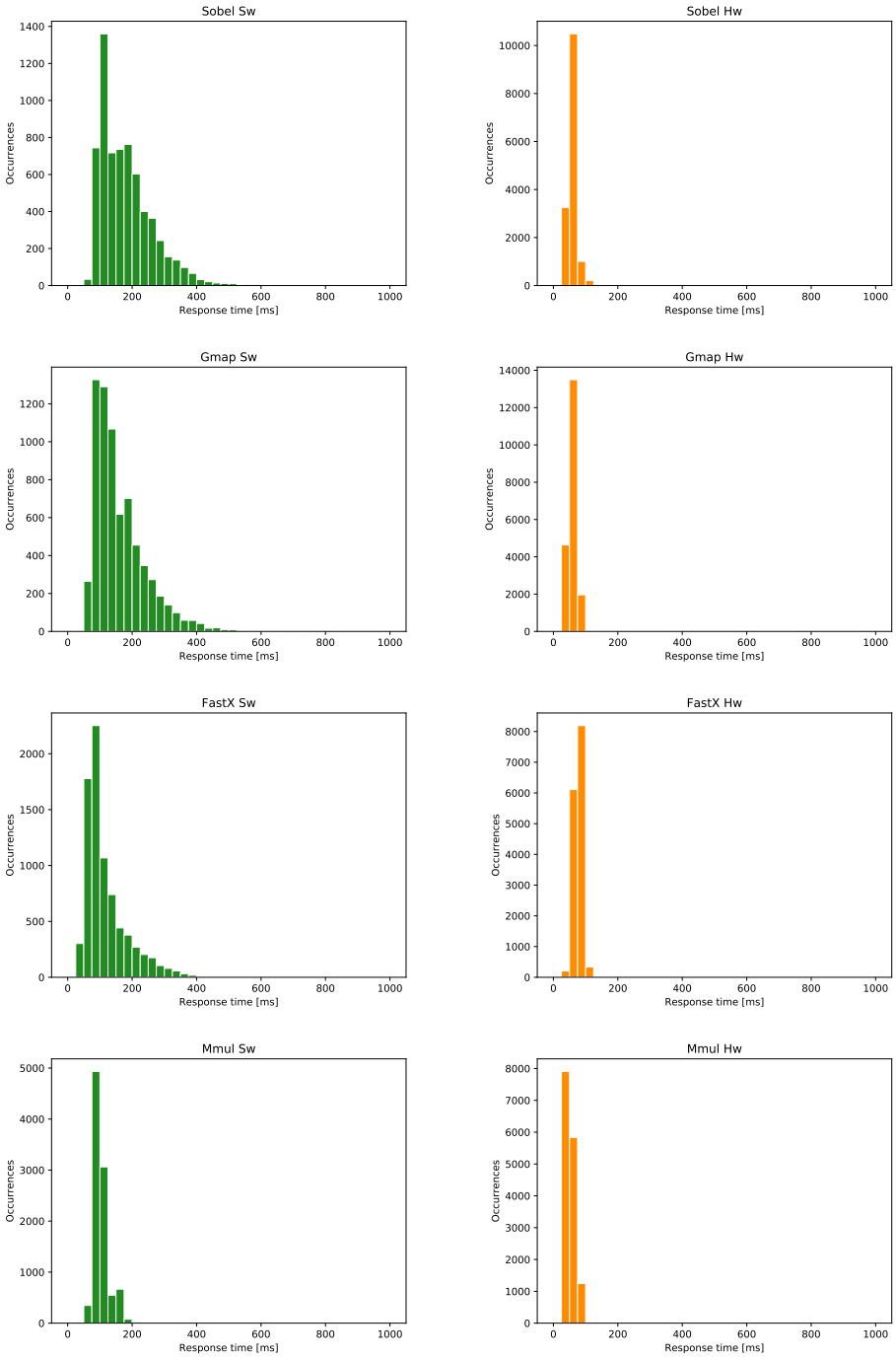

Figure 4.8: Comparison of response times distributions for SW-tasks in software and hardware mode.

Chapter 5

Enhancing BUS predictability

This chapter presents the AXI Budgeting Unit (ABU), which is a custom AXI component designed for providing bus bandwidth reservation to hardware accelerators deployed on FPGAs. An ABU shields a hardware accelerator from possible misbehaviors of other accelerators (in terms of exceeding bus data transfers) by predictably enforcing a given bus bandwidth. The ABU is not a bus arbiter but a traffic shaper component to be placed between hardware accelerators and a standard AMBA AXI bus infrastructure. ABUs can seamlessly be integrated into any FPGA design on top of the proprietary AXI Interconnect provided by vendors. This approach reduces the development costs and enhances portability and compatibility with any future releases of AXI-compliant IPs. ABUs have been implemented and tested upon SoC-FPGA platforms. After presenting a model for hardware accelerators based on the characteristics of realistic implementations (from Xilinx IP libraries and OpenCV), an analysis to bound the response times of hardware accelerators is proposed. The analysis is performed in the bus bandwidth domain and results to be tractable, as well as accurate to study FPGA-based hardware accelerators. Finally, the last part of this chapter reports a set of experimental results conducted on the Zynq-7020 aimed at demonstrating the effectiveness of the reservation mechanism implemented by ABUs, even in the presence of misbehaving hardware accelerators, and the validity of the proposed analysis.

5.1 The problem of BUS contention

Practical high-performance hardware accelerators are typically memory-intensive units capable of autonomously retrieving data from the system memory using direct memory access (DMA) or bus mastering techniques. Each HW-task accelerator is implemented using a subset of the FPGA's logic resources that are reserved only to that specific accelerator. Hence, in a setup comprising multiple hardware accelerators, their execution units operate in parallel, independently of each other, being not subject to any kind of contention regarding logic resources. In general, the response time of a hardware accelerator depends on the amount of data moved, its computation time, and the bus and memory bandwidth available from the system. The amount of data moved, and the computation time can be bounded at design time, being dependent on the accelerator design. Conversely, the bus and memory bandwidth depends on the platform's capabilities and can be a significant subject of contention with other hardware accelerators. In the context of a system comprising multiple hardware accelerators, like the one shown in Figure 5.1, bus and memory contention can become the dominant factor in determining the response time of the accelerators. If the effects of such contention are not taken into account, the interference caused and experienced by the hardware accelerators can jeopardize the predictability of the entire system.

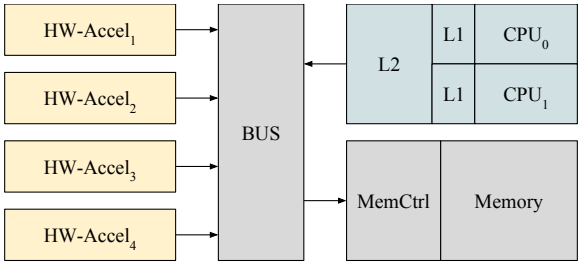

Figure 5.1: Block diagram of a custom system deployed on a SoC-FPGA platform.

This scenario is worsened by the fact that often it is not possible for a designer to control the actual bus demand rate of each accelerator deployed on the system. For instance, if the HW-task accelerator is available in the form of a closed IP, it may be impossible to tune the actual rate at which bus transactions are issued. Another aspect to consider is the increasing relevance that high-level synthesis (HLS) is gaining in the design of hardware accelerators for FPGAs [47, 16]. While these tools allow for a significant speedup of the hardware design process, they lack the precise control over the design that a register-transfer level (RTL) implementation can achieve. This effectively reduces the possibility for the designer to precisely tune the

rate of bus transactions. Finally, hardware accelerators can be plagued by design issues and bugs that may lead to execution overruns or illegal memory accesses.

To mitigate these issues, some hardware vendors typically integrate traditional priority-based arbitration in their interconnect implementations. More recent FPGA platforms also include (limited) mechanisms for QoS-aware arbitration [71]. However, the closed source nature of these implementations, often paired with an opaque description of the internals, makes it difficult to model such closed IPs and derive formal properties. In fact, the limited flexibility of those mechanisms and the lack of a proper reservation policy make them unsuited for safety-critical environments.

These challenges could be tackled by a methodology that enforces a more predictable environment, allowing for a controlled integration of first and third-party accelerators. As modern operating systems provide isolation and supervision mechanisms for software processes, it is worth providing supervision and reservation mechanisms also for the hardware activities performed by accelerators. This would enhance system predictability and enable the FPGA-based acceleration paradigm to be effectively used in safety-critical applications.

5.2 System model and Background

The proposed approach considers an AXI system composed of an interconnect, a set $\Gamma = \{\tau_1, \ldots, \tau_n\}$ of HW-tasks accelerators, and a shared sink module (e.g., a memory controller) deployed on a SoC-FPGA platform. Similarly to FRED, it is assumed that HW-task accelerators are AXI memory-mapped master modules capable of autonomously accessing data in a shared memory, which is reachable through a sink interface. Each HW-task accelerator performs the same computational activity on each run. All HW-tasks are connected to an interconnect block, which in turn is connected to the sink module.

The next subsections introduce a model for the interconnect together with some essential background related to the AXI bus, a model of the HW-tasks, and a model of the sink module. It is important to note that most of the assumptions reported in this section are only adopted for the purpose of analyzing the system (Section 5.4), while the proposed system-level mechanism (Section 5.3), i.e., the ABU, is independent of most of the adopted modeling strategies.

5.2.1 AXI interconnect

The central element of an AXI-based system is the AXI interconnect, which acts like a "switch" connecting one or more AXI master devices to one or more slave devices. The interconnect performs crucial activities such

as protocol conversions and the arbitration of bus transactions. Within the context of the system model, the interconnect is assumed to be configured in a N-to-1 mode, i.e., it connects $N \geq 1$ masters to a single slave device such as a memory controller. Under this setting, the interconnect is in charge of arbitrating the transactions issued by the master modules.

Arbitration policy

The AXI specification [6] does not mandate any specific arbitration protocol for the interconnect. Some interconnect implementations, such as the Xilinx standard Interconnect IP [8], provide two arbitration modes: (i) fixed-priority scheduling, in which the user configures static priorities for the slave ports, and (ii) a fair allocation using round-robin. In recent releases of the Vivado suite, Xilinx provides the new SmartConnect IP [64] (meant to replace the current Interconnect IP in new designs) in which the fixed-priority arbitration has been dropped retaining the round-robin arbitration only. Hence, to match realistic modern designs, this work only focuses on round-robin arbitration. In addition, it is assumed that the interconnect (i) implements *ideal* round-robin scheduling with reclaiming, i.e., the unused bandwidth is fairly re-distributed by the contenders that demand more than the fair bandwidth share, and (ii) does not introduce any overhead. Note that the actual implementation of the round-robin policy is typically not known, e.g., as it is the case of the Xilinx IPs, which are closed-source and lack of proper detailed documentation concerning arbitration policies. As a result, more accurate modeling of the arbitration may be difficult to obtain and may introduce inconsistencies among different versions of the IPs. Nevertheless, the experimental results carried out in this work surprisingly revealed a *marginal* deviation of the behavior of the Xilinx's interconnects with respect to the ideal case (see Section 5.6).

AXI Links

An AXI link provides a bidirectional connection between a master and a slave interface. Each AXI link comprises five independent transaction channels: two channels (read address and read data) for read transactions, and three channels (write address, write data, and write response) for write transactions. Each channel implements a two-way handshake mechanism by using a pair of VALID and READY signals. The producer generates the VALID signal to indicate when the address or data are available. The consumer generates the READY signal to indicate that it can accept the information. The actual transfer occurs only when both the VALID and READY signals are asserted. In order to distinguish between READY and VALID signals of read and write transactions, the letters R and W are appended before their names (e.g., RREADY and WREADY).

Read and write channels of a link can operate independently one from each other, i.e., each HW-task may perform read and write transactions concurrently. However, the AXI specification [6] does not mandate how the interconnect should manage such a level of concurrency among channel groups. In this system model, it is assumed that the interconnect arbitrates read and write channel groups independently, thus permitting concurrent read and write transactions from master modules. Please note that both the standard Interconnect and Smartconnect IPs provided by Xilinx operate in this mode [8], [64].

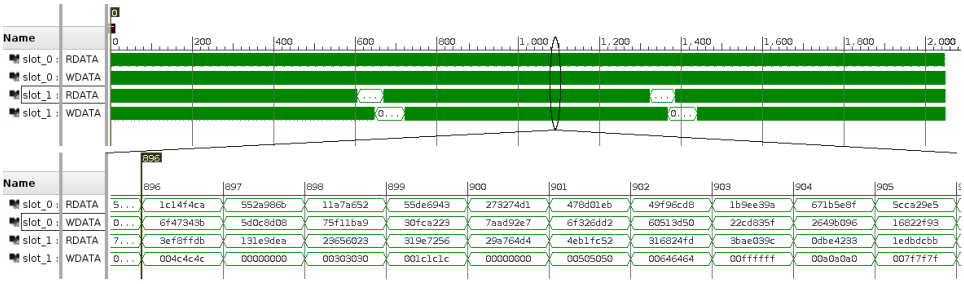

Figure 5.2: Screenshots of bus signals for read and write memory transactions of two HW-tasks (FIR and Sobel), taken from Vivado. The HW-tasks are are operating upon a Xilinx Zynq-7020 platform. The figure also reports a zoom of about 10 clock cycles.

5.2.2 HW-tasks

All HW-tasks are periodically activated, and thus generate a potentially-infinite sequence of execution instances (also referred to as jobs). Each HW-task operates like a DMA module, generating an equal number of read and write transactions. The transactions issued by each HW-task are assumed to be uniformly distributed during its execution and hence issued at a fixed rate. Please observe that despite this modeling strategy may seem coarse, many real-world hardware accelerators that perform data-parallel operations (e.g., video, image, and signal processing on raw data) present regular memory access patterns that can be modeled with a uniform demand. As a representative example, Figure 5.2 reports the bus signals for memory transactions of two state-of-the-art HW-tasks, namely a FIR filter (slot0 in the figure) and a Sobel filter from the OpenCV library (slot1 in the figure). The trace at the top of the figure reports the execution of the 0.76% and the 0.6% of a job of the two HW-tasks, respectively. The HW-tasks have been implemented with high-level synthesis (HLS) upon a Xilinx Zynq-7020 platform. As it can be noted from the figure, the FIR filter exhibits a uniform pattern of transactions (one 32-bit word per clock cycle); the same holds for the Sobel filter, with the exception of a few clock cycles every about 600 clock

cycles (the stop is attributed to the end of the processing of a row of the input image). Across all its execution, the amount of clock cycles in which the Sobel filter does not issue bus transactions corresponds to less than 10%. Nevertheless, please observe that for the purpose of analysis, the Sobel filter can still be pessimistically modeled by assuming that bus transactions are issued even in the last 600 clock cycles: further details on this strategy are discussed in Section 5.6.4.

Formally, each HW-task τ_i is characterized by the following three parameters: (i) a demand rate D_i, which represents the rate of memory transactions (both reads and writes), (ii) the maximum number N_i of memory transactions issued by each job, and (iii) its period T_i. Due to the presence of separate channels for reads and writes, the demand rate of each HW-task is bounded by two transactions per clock cycle. Demand rates are typically expressed as the number of transactions per clock cycle; when needed, a word size (such as 32-bit) may also be used in place of the number of transactions. It is very important to note that HW-tasks have very different characteristics with respect to classical software tasks. Indeed, HW-tasks have an intrinsic parallel execution and are usually implemented such that they can perform computations while issuing memory transactions (i.e., computations and memory accesses are overlapped in time). For instance, this fact can also be observed from Figure 5.2, as the hardware accelerators issue memory transactions at (almost) every clock cycle. For this reason, computations times are not modeled, and HW-tasks are assumed to be completed when they complete all their N_i memory transactions.

5.2.3 Sink module

The sink module models an endpoint block like a memory controller or a downstream AXI Interconnect (e.g., in the presence of multiple Interconnects that are connected in a hierarchical manner). Formally, the sink module is modeled with a supply bandwidth S that denotes the total rate of transactions it can accept, i.e., the maximum ratio of read and write transactions served per clock cycle.

It is worth mentioning that the size, in bytes, of a single transaction may vary even on the same system depending on how the AXI logic has been implemented on each module. Actually, the AXI standard allows connecting multiple hardware modules with different transaction word sizes or even protocol versions; the Interconnect is then responsible for converting the format of transactions. For instance, the high-performance ports included in the Zynq platforms by Xilinx for accessing the system DRAM memory dispose of a supply rate of two double-word (64-bit) transactions per clock cycle, while the default configuration of AXI master ports for hardware accelerators uses single-word transactions. When it is necessary to avoid possible inconsistencies, demand and supply rates are always expressed by

using the smallest word in the system.

5.3 AXI Budgeting Unit

This work proposes a system infrastructure that comprises a set $\mathcal{A} = \{A_1, \ldots, A_n\}$ of ABU modules controlled by a central unit named *ABU controller*. Each ABU module is conceived to be placed between a hardware accelerator and the remainder of the bus infrastructure. A sample setup is shown in Figure 5.3. The purpose of each ABU module is to *supervise* the bus traffic generated by the corresponding hardware accelerator providing both temporal and spatial isolation. Specifically, the objectives of ABUs are:

- implementing a memory bandwidth reservation mechanism by (i) keeping track of the number of bus transactions issued by HW-tasks, and (ii) enforcing a maximum *budget* of transactions within periodic time windows; and
- as a side feature, implementing a memory protection mechanism that restricts the address space accessible by HW-tasks to a set of configurable regions.

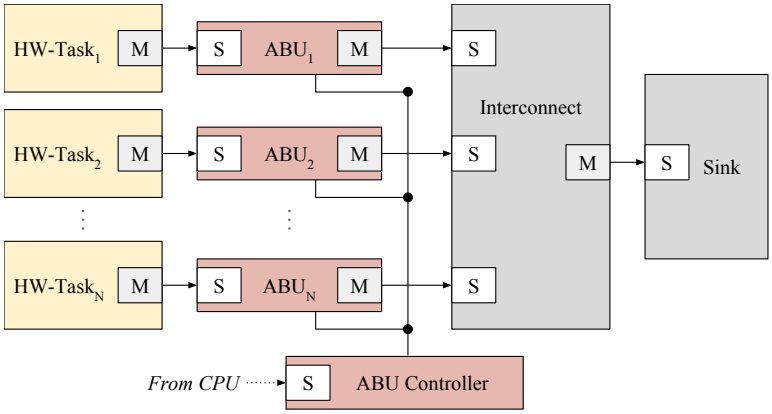

Figure 5.3: Illustration of an AXI system with hardware accelerators protected by ABUs. The boxes labeled with M and S denote master and slave AXI ports, respectively.

The ABU controller serves as a central control point that allows programming the ABU modules by means of memory-mapped registers exposed through a single AXI slave interface. In its typical usage, these memory-mapped registers are controlled by the processor (e.g., by a driver at the level of the operating system or a hypervisor). The ABU modules are, in turn, connected to the ABU controller through a custom bus, which is used to transfer configuration parameters and signals. As it is illustrated in Figure 5.3, each

ABU module also exports one AXI master and one AXI slave interface. The AXI slave interface serves as the access point for the hardware accelerator, while the AXI master port is meant to be connected to the remainder of the bus. These components are implemented in VHDL using an RTL behavioral description and deployed onto the FPGA fabric.

5.3.1 ABU working principle

According to the AXI standard, the master modules are the ones in charge of initiating bus transactions. Consequently, the HW-tasks drive the system by concurrently performing requests for bus transactions to the interconnect, which in turn selects which pending transactions need to be propagated to the sink module. The main idea behind the budgeting mechanism of ABUs is to act as a bridge between HW-tasks and the Interconnect by monitoring and altering the AXI signals. An example of an ABU in action is shown in Figure 5.4 for the case of a HW-task performing a sequence of write transactions. The figure reports the state of the AXI signals that are relevant for the considered examples, namely WVALID in output from the HW-task and the ABU (first and second rows, respectively), WREADY in output from the Sink and the ABU (third and fourth rows, respectively), and WDATA to show the data traffic on the bus (last row). The evolution of the ABU budget over time is also reported at the top of the figure. As it can be observed from the figure, when the ABU budget ends at time t_1, write transactions are blocked although the HW-task is ready to transmit data (WVALID in output from the HW-task is up) and the sink is ready to receive it (WREADY in output from the Sink is up). This is accomplished by masking signals WVALID and WREADY forcing their logic state to zero, as it is illustrated in the second and fourth rows in the figure within time interval $[t_1, t_2]$. Note that, when no budget exhaustion occurs, the ABU has a transparent behavior mirroring all signals (see time window $[t_2, t_3]$ in the figure).

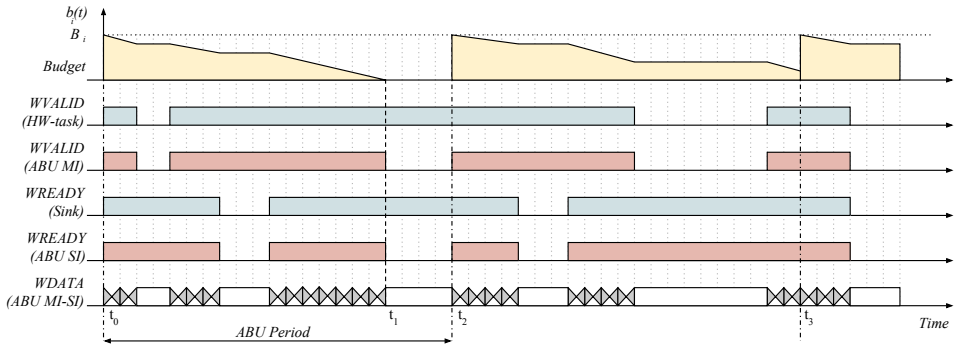

Figure 5.4: Example of ABU in action: impact on the AXI bus signals.

Budgeting mechanism

For each ABU A_i, the proposed solution allows configuring (i) a maximum budget of B_i transactions, and (ii) a period P_i with which the budget is replenished. Each ABU also keeps track of a variable parameter denoted as *instantaneous budget* b_i. At the system startup, $b_i = B_i, \forall i = 1, \ldots, n$. Then, as a HW-task performs bus transactions, the instantaneous budget is decremented until it reaches zero (budget depletion). As long as its instantaneous budget is zero, an ABU forbids bus transactions by acting on (R/W)VALID and (R/W)READY data and address signals. The instantaneous budget is recharged periodically and synchronously, i.e., if the system startup corresponds to time $t = 0$, the instantaneous budget of A_i is set to $b_i = B_i$ at every time $t = kT_i,\ k \in \mathbb{N}$. From the perspective of memory bandwidth, note that each ABU enforces a transaction rate of B_i/P_i for the corresponding HW-task *independently of the behavior of the latter.*

Memory protection

The ABU controller allows configuring X memory address regions for each ABU A_i to which the corresponding HW-task is allowed to access. Each of such regions $r_{i,j}$ (with $j = 1, \ldots, X$) is identified with a base memory address and a size, which are configurable by means of memory-mapped registers offered by the controller. Whenever a HW-task τ_i access an address outside one of the regions $r_{1,1}, \ldots, r_{i,X}$, the corresponding ABU A_i blocks all memory transactions of τ_i, as it would be disconnected from the bus; consequently, the ABU controller raises an interrupt signal. The HW-task that triggered the fault can be identified by reading a status register of the controller. The normal operation of the ABU can be restored by acting on another control register offered by the controller. This feature is particularly useful in the context of virtualized systems, where a hypervisor running on the CPU of the system-on-chip can configure the memory regions and react to illegal memory accesses.

ABU internals

The internal architecture of an ABU module is illustrated in Figure 5.5. The communication channels on the AXI link between the master and the slave interfaces are routed through a decoupler block that can stop the master from issuing transactions. The decoupler works by acting on the *ready* and *valid* signals to suspend the handshake procedure temporarily. The budgeting mechanism is implemented by means of a transaction counter that keeps track of each read/write transaction and, when the budget is exhausted, sends a signal to activate the decoupler block. The ABU controller provides a pair of registers for configuring the budget and the period of each ABU. Such registers are accessible as memory-mapped via the AXI slave interface of the

controller. The memory protection function is implemented by comparing the values on the read and write address channels with the range of addresses specified for each region $r_{i,j}$.

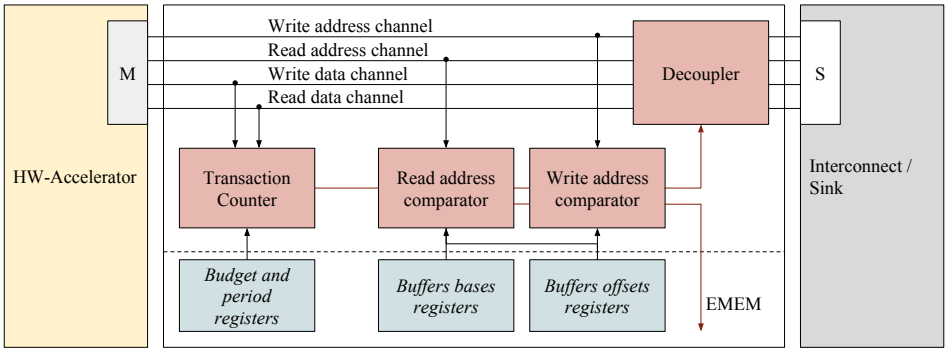

Figure 5.5: Internal functional block diagram of an ABU.

Table 5.1: Resource utilization for an ABU unit and the ABU controller on a Zynq-7020 platform.

Resource type	One ABU	ABUs Controller
LUT	436/53200 (0.82%)	279/53200 (0.56%)
FF	379/106400 (0.36%)	529/106400 (0.50%)
DSP	0/140 (0%)	0/140 (0%)
BRAM	0/220 (0%)	0/220 (0%)

Note that the core logic of ABUs is implemented with lightweight mechanisms (counters, comparators, and switches), and hence no extra clock cycles are needed to traverse ABUs. Therefore, ABUs do not introduce delays: the cost of using them is only attributed to the additional FPGA resources required to be deployed. The resource utilization of one ABU and the ABU controller, when implemented upon a Xilinx Zynq-7020 platform, is reported in Table 5.1. The table also reports the percentage of resources occupied by the two modules with respect to the total amount of resources available on the Zynq-7020. As can be noted from the table, ABUs have a very marginal impact on resource consumption.

5.4 Bandwidth-driven response-time analysis

This section studies the effect of bandwidth contention on HW-tasks under the considered modeling strategy and presents a methodology to guarantee the system predictability using the ABU. Differently from most proposals in the literature, the proposed analysis does not aim at accounting for possible interleaves of bus transactions over time (e.g., like the analysis of classical periodic real-time tasks), but aims at studying the contention incurred by

HW-tasks in the bandwidth domain, i.e., considering the actual rates at which the transactions make progress in the presence of other interfering tasks. This because, as mentioned in Section 5.2.2, real-world hardware accelerators typically perform uniformly-distributed bus transactions at a constant rate and, in particular, they even issue transactions at every clock cycle (see Figure 5.2). These characteristics make it possible to treat HW-tasks as *fluid* computational activities that make progress at a given rate (e.g., similarly to fair multiprocessor scheduling [7]), and hence allow studying the system in bandwidth domain.

In order to illustrate this peculiarity of the problem studied in this work, a simple example is firstly reported to show the effect of the contention introduced by round-robin arbitration (Sec. 5.4.1) in the bandwidth domain. Then, an observation concerning the critical instant for a set of HW-tasks is presented together with an illustrative example (Sec. 5.4.2). Finally, a strategy to enhance the system predictability by making HW-tasks prone to worst-case response-time analysis is presented (Sec. 5.5).

5.4.1 Illustrative example

To illustrate the effect of bandwidth contention incurred by HW-tasks subject to round-robin arbitration, consider a system composed of (i) a sink module providing a supply of $S = 6$, (ii) an interconnect directly connected to the sink module, and (iii) three HW-tasks, namely τ_1, τ_2, and τ_3, directly connected to the interconnect. The HW-tasks have the same demand $D_1 = D_2 = D_3 = S/2 = 3$ corresponding to half of the supply. The first HW-task (τ_1) needs to perform $N_1 = 6$ transactions and has a period of $T_1 = 9$ time units. The second HW-task (τ_2) performs $N_2 = 24$ transactions within a period of $T_2 = 11$ time units. Finally, the third HW-task (τ_3) performs $N_3 = 30$ transactions within a period of $T_3 = 15$ time units.

To avoid possible misunderstanding, please bear in mind that HW-tasks are statically allocated onto the FPGA area and hence do not contend the logical resources of the FPGA. For this reason, HW-tasks operate in a *parallel* fashion using their own (private) logic resources and can incur in contention only when issuing bus transactions.

Consider the case in which all HW-tasks are synchronously released at the same instant $t = 0$. Figure 5.6(a) illustrates the resulting schedule of the three HW-tasks by showing the intervals of time in which they are operating (on the top of the figure) and the repartition of the bandwidth over time (on the bottom of the figure). Each square unit of the bandwidth supply in the figure represents a transaction unit. At time $t = 0$, since the total bandwidth demanded by all HW-tasks $D_1 + D_2 + D_3 = 9$ exceeds the available bandwidth supply $S = 6$, the Interconnect limits the bandwidth of the three HW-tasks to a fair share of $S/3 = 2$. This bandwidth allocation continues up to $t = N_1/(S/3) = 3$, when τ_1 finishes its execution. Once τ_1 completes,

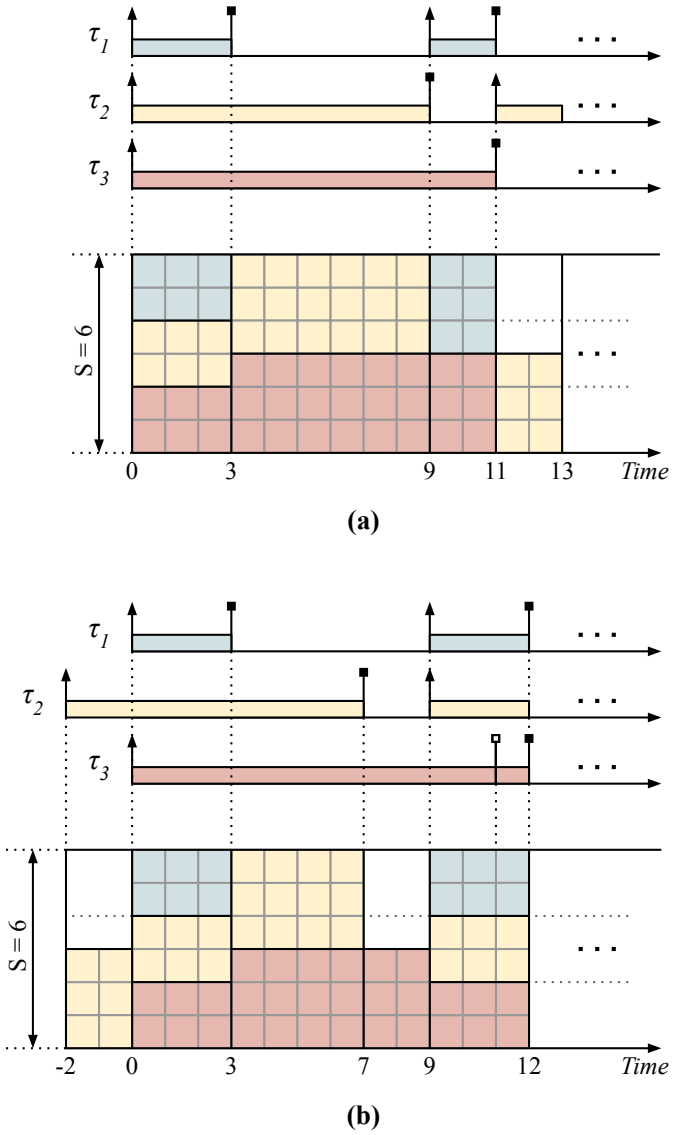

Figure 5.6: Examples of HW-task scheduling in the bandwidth domain with (a) synchronous release and (b) without synchronous release. In (b), HW-task τ_3 experiences a longer response time with respect to the schedule in (a).

τ_2 and τ_3 can proceed at their full rate of $S/2 = 3$ without suffering any contention. At time $t = 9$, τ_2 completes but a new periodic instance of τ_1 is also released. Again, both τ_1 and τ_3 can progress at their full rate without contention. At time $t = 11$, τ_1 and τ_3 complete at the same time and a new instance of τ_2 is activated. The latter can then proceed to operate while no other HW-task is active. Since τ_2 demands a bandwidth of $D_2 = 3$, half of the supply is left unused up to the next activation of τ_3 (which will occur at time $t = 15$).

5.4.2 Analysis issues

As it can be noted from Figure 5.6(a), HW-tasks are "slowed down" only when the total bandwidth demanded by active HW-tasks exceeds the supply (as it happens in $[0, 3)$ in the figure), i.e., when they make progress at a rate that is lower than their demand. Clearly, this phenomenon affects the worst-case response times of the HW-tasks.

Unfortunately, a bandwidth-driven response-time analysis cannot be accomplished by leveraging classical techniques for periodic real-time tasks. In particular, when studying the problem, we identified a set of issues (in some way similar to those identified in the analysis of multiprocessor real-time systems under global scheduling [22]) that prevent to analyze the system by looking at a single scheduling scenario.

To provide a taste of the identified issues, this section demonstrates that the classical critical instant theorem for periodic real-time tasks under uniprocessor scheduling does not hold for the problem studied in this work. Indeed, the longest response time of a HW-task may not occur when it is synchronously released together with all other HW-tasks.

To this end, consider the same system setup used for the previous example (Sec. 5.4.1). This time, assume that τ_2 is released before τ_1 and τ_3 at time $t = -2$, as shown in Figure 5.6(b). In this way, the first job of τ_2 can issue six transactions without suffering contention before τ_1 and τ_3 are activated at time $t = 0$. Hence the first job of τ_2 completes early (time $t = 7$) with respect to the case of synchronous release, leaving half of the bandwidth supply unused in time interval $[7, 9)$. Since τ_2 has been released earlier, also its next instance will be released earlier at time $t = 9$. Such a second job of τ_2 interferes with both τ_1 and τ_3 causing τ_3 to finish at time $t = 12$, i.e., one unit of time later than in the case of synchronous release. In conclusion, by releasing τ_2 two unit of time earlier with respect to the synchronous release case, the response time of τ_3 increases by one unit of time.

Proving a correct critical instant for the general case resulted in a challenging problem that is still open. Nevertheless, as it is shown in the following section, ABUs can be extremely useful to make the system far more prone to analysis, hence increasing its predictability.

5.5 Response-time analysis with ABUs

Besides ABUs implement resource reservation, hence protecting the system from misbehaving HW-tasks, they can also be leveraged at the stage of analysis to help bounding the response times of the HW-tasks. Indeed, under the assumption that the ABU periods are orders of magnitude smaller than the periods of the HW-tasks, i.e., $P_i \ll \min_{i=1,\dots,n} \{T_i\}$, ABUs can act as bandwidth regulators limiting the maximum demand rate of HW-tasks.

Differently to software-based reservation techniques, for which a short reservation period determines a high overhead, the assumption on ABUs' periods is practical because ABUs are realized in hardware and hence do not introduce relevant issues when adopted with short reservation periods. Specifically, as mentioned in Section 5.3, ABUs are designed is such a way that do not introduce delays and do not represent bottlenecks for the logic circuits deployed onto the FPGA such that the operating frequency of the latter has to be limited.

Under this setting, each ABU offers to the corresponding HW-task a virtual, dedicated supply of bus bandwidth B_i/P_i, which is independent of the behavior of the other HW-tasks as long as the ABU budgets are guaranteed. Therefore, the problem of analyzing a set of HW-tasks protected by ABUs can be decomposed into two independent steps:

1. guaranteeing that a set of ABUs can provide the corresponding bandwidths in the worst case, i.e., their entire budgets can be safely provided in every period; and

2. guaranteeing that the bandwidth provided by each ABU is sufficient for the corresponding HW-task to meet its deadline.

These steps are addressed in the following two sub-sections, respectively.

5.5.1 Analyzing ABUs

As long as the sum of the bandwidths provided by a set of ABUs does not exceed the total supply S, i.e., $\sum_{i=1}^{n} B_i/P_i \leq S$, no contention can occur; therefore, it is guaranteed that their budgets can be provided within every periodic instance. However, in the general case, this condition may not hold, and hence the analysis of ABUs must account for contention exactly as discussed in the example of Section 5.4.1.

Nevertheless, differently from a direct analysis of HW-tasks, two observations can be leveraged to make the analysis of ABUs tractable. First, as mentioned in Section 5.3, ABUs are synchronously activated at the system startup. Second, due to the assumption on the ABUs' periods ($P_i \ll \min_{i=1,\dots,n} \{T_i\}$), there is no particular advantage in assigning heterogeneous periods to ABUs, and hence to act as fluid bandwidth regulators they can be all configured

with the same period P. How to configure a suitable value for the period P is discussed in the experimental evaluation reported in Section 5.6.

Under this setting, it is then sufficient to study the case of synchronously released ABUs by analyzing a single problem window of length P that contains a single periodic instance of each ABU. In other words, it is enough to verify that all ABUs can provide their budget before time $t = P$ assuming that they are all released at time $t = 0$.

When contention occurs, it is not straightforward to compute how the available bandwidth supply is distributed between a set of active HW-tasks. In fact, considering n arbitrary HW-tasks and a supply S, they can be classified in (i) those that demand less (or the same) bandwidth than the fair share S/n, and (ii) those that demand more bandwidth than S/n, with the result that the spare bandwidth left by HW-tasks of type (i) is fairly re-distributed between the HW-tasks of type (ii). Algorithm 1 is presented to account for this phenomenon and computes the actual share of bandwidth of a supply S for each HW-task in a set $\mathcal{C}_{HW}$ of contending HW-tasks.

Algorithm 1: Computing bandwidth shares.

Input: *A set of HW-tasks:* $\mathcal{C}_{HW} = \{\tau_1, \ldots, \tau_m\}$
Input: *Sink supply:* S
Output: *A set of bandwidth shares:* $\overline{\mathcal{D}} = \{\overline{D}_1, \ldots, \overline{D}_m\}$

```
1  begin
2      S_rem ← S
3      M ← |C_HW|
4      for τ_i^hw ∈ C_HW by increasing D_i do
5          D̄_i ← min(D_i, S_rem/M)
6          S_rem ← S_rem − D̄_i
7          M ← M − 1
8      end
9      return D̄
10 end
```

The correctness of the algorithm is stated by the following lemma.

Lemma 1. *Given a sink with supply S and a set of HW-tasks $\mathcal{C}_{HW}$ that contend for the supply, Algorithm 1 computes the actual share of bandwidth $\overline{D}_i$ assigned to each HW-task $\tau_i \in \mathcal{C}_{HW}$ under a fair arbitration policy which evenly distributes the available bandwidth among the HW-tasks.*

Proof. The proof is by induction on the iterative steps of the algorithm. *Base case (first iteration, $M = |\mathcal{C}_{HW}|$):* Let τ_i be the HW-task considered at the first iteration. If $D_i \geq S/M$, then by line 5 τ_i is assigned a bandwidth share $\overline{D}_i = S/M$, which is correct, as it corresponds to the fair share. Since the set of HW-tasks is explored in order of increasing D_i (see line 4), then all the following iterations will consider HW-tasks with $D_i \geq S/M$ and,

for the same reason, will be assigned a bandwidth equal to the fair share. Otherwise, if $D_i < S/M$, then the HW-task will be assigned a bandwidth share equal to the required demand $\overline{D}_i = D_i$. Note that this cannot affect the bandwidth assignment of the other HW-tasks as D_i is lower than the fair share S/M. *Inductive case ($M < |\mathcal{C}_{HW}|$):* Suppose that the algorithm assigned a correct bandwidth to the first $|\mathcal{C}_{HW}| - M + 1$ HW-tasks and that it remains to distribute a supply bandwidth S_{rem} to $M < |\mathcal{C}_{HW}|$ HW-tasks. Let τ_i be the HW-task considered at the current iteration. Similarly to the base case, if $D_i \geq S_{rem}/M$, then by line 5 τ_i is assigned a bandwidth share $\overline{D}_i = S_{rem}/M$, which is correct, as it corresponds to the fair share with respect to the remaining $M - 1$ HW-tasks. Again, since the set of HW-tasks is explored in order of increasing D_i, the same will hold for all the following iterations. Otherwise, if $D_i < S_{rem}/M$, then the HW-task will be assigned a bandwidth share equal to the required demand $\overline{D}_i = D_i$, which again cannot affect a fair distribution for the following $M - 1$ HW-tasks. Hence the lemma follows. □

Leveraging Algorithm 1, it is finally possible to build a schedulability test that verifies whether a set of ABUs can provide their budget within their period P. This is accomplished by Algorithm 2, which unrolls the execution of a set of HW-tasks protected by ABUs within an analysis window $[0, P]$.

The algorithm inputs the set of HW-tasks $\mathcal{T}_{HW}$ and the corresponding set of ABUs $\mathcal{A}$ (the i-th ABU is connected to the i-th HW-task), and returns a boolean predicate that indicates whether the ABUs are schedulable or not. The algorithm keeps track of the analysis time t (initialized to $t = 0$) and the instantaneous budget b_i available for each ABU A_i, which is initialized to B_i (full budget). At the beginning of the ABU period ($t = 0$), all ABUs have available budget and hence all HW-tasks are considered active, i.e., they can generate transactions. Consequently, at line 4, the set of active HW-tasks, denoted with $\mathcal{C}_{HW}$, is initialized to $\mathcal{T}_{HW}$. Then, the procedure enters a loop at line 5. At each iteration, the algorithm computes the distribution of the supply S among the active HW-tasks by means of Algorithm 1, so obtaining the share of bandwidth $\overline{D}_i$ for each HW-task $\tau_i \in \mathcal{C}_{HW}$. Subsequently, it computes the amount of time Δ needed by at least one ABU A_i to provide all the available budget b_i, which is given by $\Delta = \min(b_i \,/\, \overline{D}_i)$. If a HW-task is not able to complete within the period P, then the system is deemed unschedulable and the algorithm terminates (lines 8-9). Otherwise, the algorithm proceeds by updating the budget of each ABU accounting for a lower-bound on the transactions performed in an interval of length Δ (line 12). Also, if the budget of an ABU is depleted ($b_i = 0$), then the corresponding HW-task is prevented to issue transactions and hence is removed from the set of active HW-tasks $\mathcal{C}_{HW}$ (line 14).Finally, the algorithm advances the time t by Δ and continues to iterate until the set $\mathcal{C}_{HW}$ is empty. If the algorithm completes without never detecting a deadline miss at lines 8-9, then the

Algorithm 2: Analysis of ABUs.

Input: *A set of HW-tasks:* $\mathcal{T}_{HW} = \{\tau_0, \ldots, \tau_n\}$
Input: *A set of ABUs:* $\mathcal{A} = \{A_0, \ldots, A_n\}$
Output: *Result of the schedulability test (true/false)*

```
1  begin
2      t ← 0
3      b_i ← B_i  ∀i = 1, ..., n
4      C_HW ← T_HW
5      while C_HW ≠ ∅ do
6          D̄ ← Algorithm 1(C_HW, S)
7          Δ ← min_{τ_i ∈ C_HW} (b_i / D̄_i)
8          if Δ + t ≥ P then
9              return false
10         end
11         for τ_i ∈ C_HW do
12             b_i ← b_i − ⌊D̄_i · Δ⌋
13             if b_i = 0 then
14                 C_HW ← C_HW \ {τ_i}
15             end
16         end
17         t ← t + Δ
18     end
19     return true
20 end
```

system is deemed schedulable. Finally, the following lemma states that the analysis of Algorithm 2 is sustainable, i.e., increasing the ABU budgets can only worsen the schedulability of a set of ABUs (and, vice versa, a set of schedulable ABUs remains schedulable if the budgets are decreased).

Lemma 2. *The schedulabiliy test provided by Algorithm 2 is sustainable with respect to budgets* B_i*.*

Proof. Suppose that a set of ABUs is not schedulable according to Algorithm 2. Hence, there exists a certain time t at which the condition at line 8 holds. Consider an arbitrary ABU A_i (associated to task τ_i) and let $[0, t')$ be the interval of time in which τ_i is in set $\mathcal{C}_{HW}$ during $[0, t)$, i.e., $t' \leq t$. There are two cases: (i) τ_i is still in set $\mathcal{C}_{HW}$ at time t (i.e., $t' = t$), (ii) τ_i left set $\mathcal{C}_{HW}$ before time t (i.e., at time $t' < t$).

Case (i): In $[0, t)$, τ_i always contributed to the bandwidth distribution by means of Algorithm 1. Hence, if the budget B_i is increased, the bandwidth shares $\overline{D}_i$ assigned during $[0, t)$ are the same and therefore the schedulability result cannot change. If $\Delta = b_i/\overline{D}_i$ (i.e., at time t, τ_i is the task detected to miss its deadline), then, by increasing the budget B_i, Δ can only increase and hence the condition at line 8 would hold too.

Case (ii): Similarly to the previous case, τ_i always contributed to the bandwidth distribution in $[0, t')$ and hence, if B_i is increased, the execution of Algorithm 2 cannot change up to time t'. If the budget B_i is increased to $B_i + \varepsilon$, at time t' it can be either that the value of Δ remains the same, or that it increases too by ε. Consequently, τ_i will remain for more time into set $\mathcal{C}_{HW}$, contributing to the bandwidth distribution also after time t', or still leaves set $\mathcal{C}_{HW}$ at time t'. In both these cases the schedulability result cannot change.

Hence the lemma follows. □

5.5.2 Analysis example

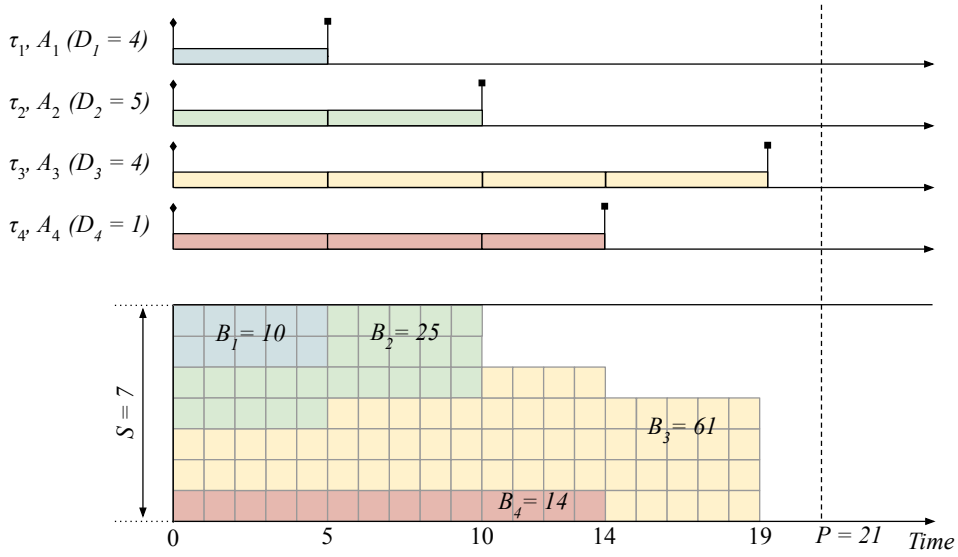

Figure 5.7: Example of HW-tasks executing within an ABU period.

Figure 5.7 presents an example of the schedulability test presented in Algorithm 2 using a system composed of (i) a sink module providing a supply of $S = 7$, (ii) an interconnect directly connected to the sink module, and (iii) four HW-tasks, formally $\mathcal{T}_{HW} = \{\tau_1, \tau_2, \tau_3, \tau_4\}$. The HW-tasks are connected to the interconnect through their respective ABUs, A_1, A_2, A_3, and A_4. The first HW-task (τ_1) has a demand of $D_1 = 4$ and an ABU budget $B_1 = 10$. The second HW-task (τ_2) has a demand $D_2 = 5$ and an ABU budget of $B_2 = 25$. The third HW-task τ_3 has a demand of $D_3 = 4$ and an ABU budget $B_3 = 61$. Finally, the last HW-task τ_4 has a demand of $D_4 = 1$ and an ABU budget of $B_4 = 14$. All ABUs $A_1, \ldots, A_4$ are configured with the same period $P = 21$.

At the beginning of the ABU period ($t = 0$), all HW-tasks are simultaneously activated ($\mathcal{C}_{HW} \leftarrow \mathcal{T}_{HW}$) since the ABUs' instantaneous budgets have

been replenished to their full capacity: $b_1 = 10$, $b_2 = 25$, $b_3 = 61$, and $b_4 = 14$. First, the supply shares $\overline{D}_i$ are computed using Algorithm 1 resulting in $\overline{D}_1 = 2$, $\overline{D}_2 = 2$, $\overline{D}_3 = 2$, and $\overline{D}_4 = 1$. Then, the interval Δ is computed as $\Delta = \min(b_i \,/\, \overline{D}_i) = b_1 \,/\, \overline{D}_1 = 5$. Since the sum of current time $t = 0$ and $\Delta = 5$ is less than the ABU period $P = 21$, the algorithm can continue by (i) updating the instantaneous budgets $b_1 = 0$, $b_2 = 15$, $b_3 = 51$, and $b_4 = 9$; (ii) removing τ_1 from the set of active HW-task ($\mathcal{C}_{HW} \leftarrow \mathcal{C}_{HW} \setminus \{\tau_1\}$) since it has exhausted his ABU budget; and (iii) updating the time $t = t + \Delta = 5$. In the next iteration, the new bandwidth shares are computed as $\overline{D}_2 = 3$, $\overline{D}_3 = 3$, and $\overline{D}_4 = 1$. Then, the new interval is computed as $\Delta = b_2 \,/\, \overline{D}_2 = 5$. Consequently, (i) the instantaneous budgets are updated $b_2 = 0$, $b_3 = 36$, and $b_4 = 4$; (ii) τ_2 is removed form the set of active HW-tasks ($\mathcal{C}_{HW} \leftarrow \mathcal{C}_{HW} \setminus \{\tau_2\}$) having exhausted its budget; and (iii) time is updated $t = t + \Delta = 10$. In the next iteration, only τ_3 and τ_4 are active. Hence, the new bandwidth shares are computed as $\overline{D}_3 = 4$ and $\overline{D}_4 = 1$, which corresponds to their demands $\overline{D}_3 = D_3$, $\overline{D}_4 = D_4$ since their sum is less that the supply $S = 7$. The new interval is computed as $\Delta = b_4 \,/\, \overline{D}_4 = 4$. Consequently, (i) the instantaneous budgets are updated $b_4 = 0$ and $b_3 = 20$; (ii) τ_4 is removed form the set of active HW-tasks ($\mathcal{C}_{HW} \leftarrow \mathcal{C}_{HW} \setminus \{\tau_4\}$) having exhausted its budget; and (iii) time is updated $t = t + \Delta = 14$. Finally, in the last iteration, the bandwidth shares of τ_3 corresponds to its demand $\overline{D}_3 = D_3 = 4$. Hence, τ_3 depletes its budget at $\Delta = b_3 \,/\, \overline{D}_3 = 5$, and the time can be updated to $t = t + \Delta = 19$. Since $t \leq P$, all HW-task are able of exhausting their budgets within the period. Hence, the system is feasible.

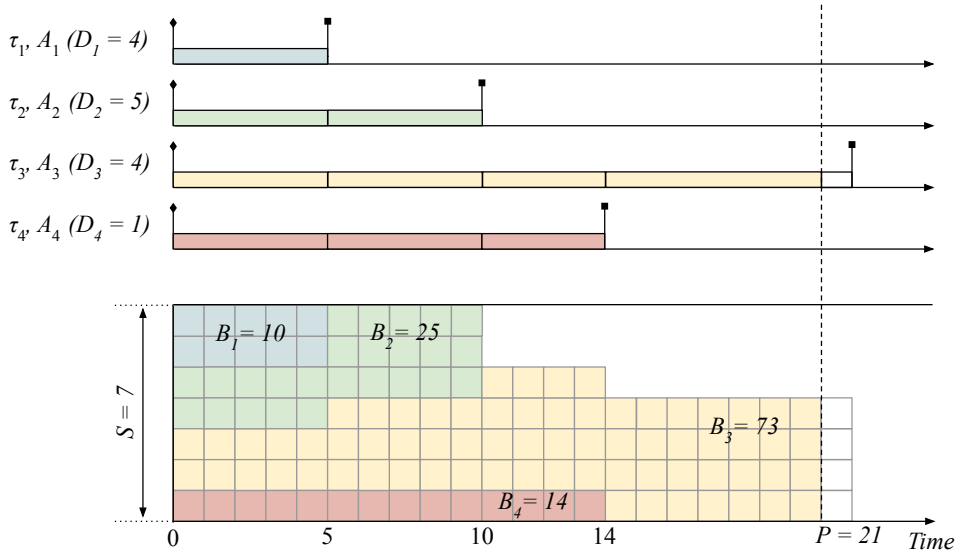

Figure 5.8: Example of HW-tasks executing within an ABU period with a budget overrun.

Figure 5.8 presents the same scheduling example with the only difference that the ABU budget of τ_3 has been increased to $B_3 = 73$. In this case, the system becomes unfeasible since τ_3 it is unable of exhausting his budget within a single ABU period. Please note that this happens even if the sum of the ABU budgets is smaller than the sink capacity within an ABU period $\sum B_i = 122 \leq P \cdot S = 147$, because τ_3 is unable to fully utilize the available supply due to its limited demand.

5.5.3 Assigning ABU budgets

As ABUs act as bandwidth regulators for HW-tasks, they enforce a specific rate at which transactions are issued. Specifically, a HW-task τ_i protected by ABU A_i issues transactions at rate B_i/P as long as the ABU is guaranteed to be schedulable according to the analysis presented in the previous section. Therefore, to guarantee that τ_i is capable of performing N_i transactions within its implicit deadline T_i, it is sufficient that the following inequality is satisfied: $\frac{N_i}{B_i/P} \leq T_i$. By rewriting the latter equation, it is possible to derive a constraint on the ABU budgets to ensure the schedulability of a set $\mathcal{T}_{HW}$ of HW-tasks, i.e.,

$$\forall \tau_i \in \mathcal{T}_{HW}, \quad B_i \geq \frac{N_i \cdot P}{T_i}. \tag{5.1}$$

Note that the same constraint can be generalized to the case of constrained deadlines by simply replacing T_i with the relative deadline of the HW-task.

Lemma 3. *If a set of HW-tasks $\mathcal{T}_{HW} = \{\tau_0, \ldots, \tau_n\}$ respectively protected by a set of ABUs $\mathcal{A} = \{A_0, \ldots, A_n\}$ is not schedulable (according to Algorithm 2) by setting the ABU budgets as $B_i = \frac{N_i \cdot P}{T_i}$, then it is not schedulable with any other budget assignment.*

Proof. Given the constraint of Equation (5.1), $B_i = \frac{N_i \cdot P}{T_i}$ is the *minimum* budget for each ABU A_i such that the schedulability of τ_i can be guaranteed. Hence, feasible budget configurations can include only budget values larger than $\frac{N_i \cdot P}{T_i}$. By Lemma 2, if a set of ABUs is not schedulable by assigning such minimum budgets, then it is also not schedulable by assigning larger budgets. Hence the lemma follows. □

5.6 Experimental evaluation

To assess the effectiveness of the ABUs on a real hardware system, an experimental evaluation has been conducted on the Zynq-7020 SoC platform by Xilinx. The experimental evaluation is structured in two parts: the first part aims at evaluating the effectiveness of the reservation mechanism enforced by the ABUs using DMA-like HW-tasks; the second part evaluates the ABUs with a case study application that comprises a finite impulse

response (FIR) HW-task for signal processing and a Sobel HW-task for image processing from OpenCV.

All HW-tasks used in this evaluation have been designed with Xilinx's Vivado HLS. The choice of utilizing HLS comes from the steadily increasing relevance that high-level synthesis is assuming in the design of hardware accelerators. For instance, a HLS tool can also be used to synthesize a HW-task implementing a custom compute unit for executing an OpenCL kernel. The hardware-level interface of the HW-tasks used in this evaluation consists of (i) two AXI4 master interfaces for accessing the system memory; (ii) one AXI4-lite slave control interface, to expose a set of memory-mapped registers through which the software can control the HW-task; and (iii) an interrupt signal to notify the processor when the computation of the HW-task is completed.

Each HW-task is controlled by a periodic software task running on top of the FreeRTOS kernel, which in turn runs upon one the Cortex-A processors of the Zynq-7020. The software task relies on a device driver for managing the HW-task, feeding the addresses of the source and destination memory buffers as arguments. The driver controls the HW-tasks through the set of control registers exported via the AXI4-lite slave interface. Each job of each software task starts the corresponding HW-task and then self-suspends waiting for the HW-task to complete the execution. When the HW-task has completed, it sends an interrupt signal, which is caught by the interrupt service routine included in the driver. The service routine, in turn, wakes up the software task, which can then complete its job. This evaluation is focused on the timing properties of HW-tasks only.

5.6.1 Evaluation of the reservation mechanism

The first part of the experimental evaluation aims at validating the effectiveness of the reservation mechanism when one or more HW-tasks deviate from the nominal behavior by demanding a higher transaction rate and issuing more transactions than expected. Note that, from the perspective of bus contention, the bus transactions issued by HW-tasks are the only relevant aspect. Therefore, this evaluation employs a set of DMA-like HW-tasks, which allows for an almost-arbitrary control of the bus transactions that are generated. Nevertheless, also note that several hardware accelerators for FPGAs, including those of the Xilinx's IPs library such as FFT [25], FIR filter [26], and Convolution Encoder [18], require the support of a DMA for accessing the system memory.

Variants of HW-tasks

To simulate the effect of a misbehaving HW-task, three variants of the same DMA-like HW-task have been designed. Each variant differs by the

amount of data N_i and the demand rate D_i. The parameters of these variants, referred to as modes, are summarized in Table 5.2. The demand value in MB/s is calculated by considering that each bus transaction involves a 32-bit word and that the clock rate of the FPGA is set to 100 MHz. All the HW-tasks issue 16-word burst transactions. On the Zynq-7020, the maximum supply bandwidth S available to access the memory from the PS through a HP port is four transactions per clock for each port, as they operate in 64-bit mode (the DRAM clock is set to 525 MHz).

Table 5.2: Configuration of HW-tasks. The demand D_i is expressed in both transactions per clock cycle and in megabytes per second.

HW-task mode	D_i		N_i	
	[tr/clk]	[MB/s]	[tr]	[MB]
1	2	763	524288	2
2	1	381	262144	1
3	2/3	254	131072	0.5

Description of the experimental setting.

The system setup used for this evaluation comprises four DMA-like HW-tasks allocated on the Zynq's PL and connected to a single HP port through an AXI Interconnect. The Interconnect is set in performance mode to maximize the bandwidth available to the HP port. An ABU module is placed between each HW-task and the Interconnect. The baseline configuration includes two HW-tasks, τ_1 and τ_2, set in mode 1, a HW-task, τ_3, operating in mode 2, and the last HW-task τ_4 set in mode 3. This configuration represents the system operating in *nominal conditions*, i.e., when all the HW-tasks respect their nominal demand D_i and data length N_i values, and is referred to as *nom*. To study the effect of misbehaving HW-tasks, two additional variants of the baseline configuration have been defined. In the first misbehaving configuration, referred to as *misb-3*, τ_3 operates in mode 1 instead of mode 2. This configuration, represents the case in which a single HW-task exceeds its nominal values, demanding a higher transaction rate and length. In the second misbehaving configuration, named *misb-3-4*, τ_3 and τ_4, normally operating in mode 2 and mode 3 respectively, now operate in mode 1. This configuration aims at reproducing the scenario in which two HW-tasks exceed their nominal values.

5.6.2 Profiling HW-tasks

The first set of experiments has been carried out to characterize the system configurations without ABUs. To this end, a separate profiling experiment has been conducted for each configuration of the system: the base configuration *nom*, and two misbehaving configurations *misb-3* and

misb-3-4. These experiments allow evaluating the impact of one or more misbehaving HW-tasks on the response times of the other HW-tasks when using the default round-robin arbitration policy of the Interconnect. For this set of experiments, τ_1 is activated every 10 ms, τ_2 every 15 ms, τ_3 every 25 ms, and τ_4 every 50 ms. Measurements on the hardware have been conducted with multiple runs by testing random activation offsets of the HW-tasks, for a total of about 30 minutes of execution (collecting data for hundreds of thousands of jobs). Figure 5.9 presents the results of these experiments by reporting the longest-observed response times on the real hardware as solid color bars. The results corresponding to the misbehaving HW-tasks are highlighted with different colors and patterns. Comparing the response times observed under nominal conditions (*nom*) with the response times obtained under misbehaving configurations, it is evident that even a single misbehaving HW-task (*misb-3*) could have a significant impact on the response time of the other HW-tasks. This effect becomes even more tangible when taking into account the configuration *misb-3-4* in which two HW-tasks misbehave. For instance, the response time of τ_1 in *misb-3-4* increases by more than 50% with respect to nominal conditions.

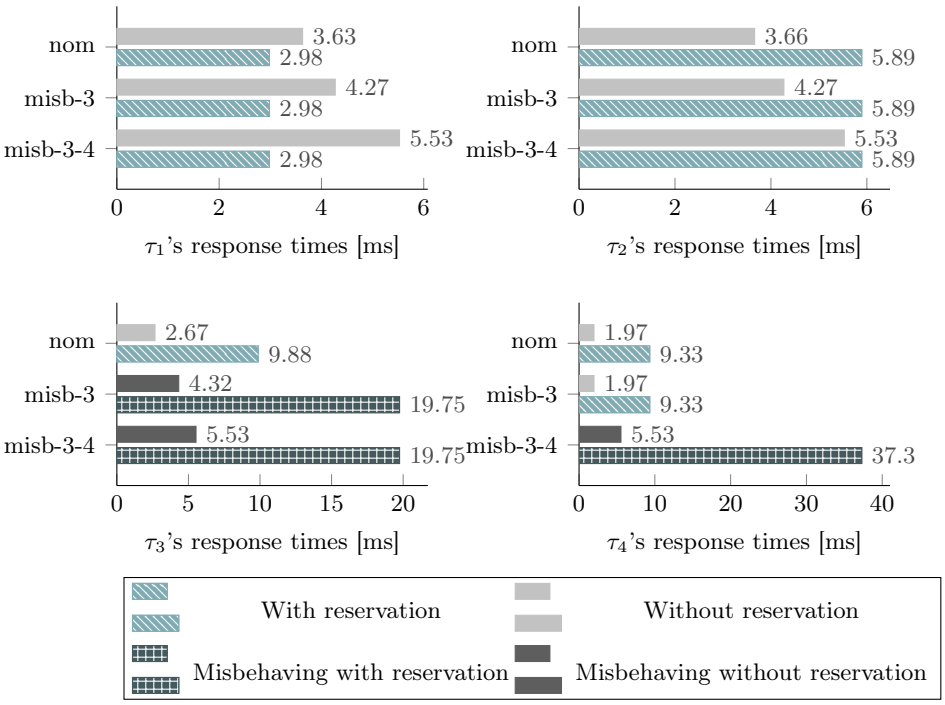

Figure 5.9: Response times of four HW-tasks without and with ABUs under multiple configurations.

5.6.3 Evaluating the reservation mechanism

The following set of experiments analyzes what happens when the ABUs are present. These experiments serve two purposes: first, to test the effectiveness of temporal isolation between HW-tasks; second, to confirm that the assumptions made in Sections 5.2 and 5.4 to model and analyze the system are realistic. To this end, the longest-observed response times on the hardware have been compared with the response-time bounds computed by the analysis of Section 5.4. The ABUs have been configured according to the minimum budgets provided by Lemma 3 under nominal conditions. The period of ABUs has been selected according to the following rationale. Since the ABUs count integer transactions, the period must be chosen as the smallest value that can ensure that all the minimum budgets provided by Lemma 3 are integers. Furthermore, to avoid splitting transaction bursts, it is worth choosing a period such that the budget is a multiple of the burst size. Such a period can be easily obtained with a binary search. The resulting ABU configuration for this experimental setting is reported in Table 5.3. The table also reports the response times, both observed on the hardware and obtained by the analysis proposed in this work, under configuration *nom*.

As it can be noted from Figure 5.9, ABUs allow controlling the longest-observed response times (e.g., fixed to 2.98 for τ_1) independently of the behavior of the other HW-tasks; indeed, the response times are the same even in the misbehaving configurations *misb-3* and *misb-3-4*. Clearly, this improvement is achieved at the expenses of the misbehaving tasks (τ_3 and τ_4): in fact, their response times in misbehaving configurations is penalized.

Table 5.3: Configuration parameters for the ABUs and response times for the corresponding HW-tasks under the nominal configuration.

HW-Task	ABU		Response times [ms]	
	B_i [tr]	P [clk]	Longest observed	By analysis
τ_1	224	128	2.982	2.995
τ_2	112		5.893	5.991
τ_3	32		9.876	10.485
τ_4	16		9.328	10.485

5.6.4 A case study

The second part of the experimental evaluation considers a case-study application that comprises a FIR filter HW-task for signal processing, a Sobel HW-task for image processing, and two DMA-like HW-tasks operating in mode 1. The FIR filter implements a 12th order low-pass filter designed to process 16kHz audio samples with a cutoff frequency of 4 kHz. Internally, the FIR filter uses fixed-point representations to take advantage of the FPGA's DSP blocks. Each instance of the FIR filter processes 1 MB of samples.

The Sobel filter processes 640 × 480 RGB images with 24-bit color depth, resulting in a size of 1200 KB. Table 5.4 summarizes the characteristics of these accelerators, which both issue 16-word burst transactions.

Table 5.4: Parameters of the Sobel and FIR hardware accelerators.

HW-task	D_i		N_i	
	[tr/clk]	[MB/s]	[tr]	[KB]
Sobel	1.9	725	614400	2400
FIR	2	763	524288	2048

As visible from the trace shown in Figure 5.2, the access pattern generated by the Sobel filter HW-task is not strictly uniform due to a short pause occurring between two image lines. Such a signal analysis has been performed on the real hardware by instrumenting the design with an integrated logic analyzer (ILA) module. Clearly, the access pattern of the Sobel HW-task violates the uniform transaction hypothesis made in Section 5.2 to model the system. However, by performing the pessimistic assumption that the Sobel HW-task continues issuing transactions even during the brief pause between a line and the next, it is still possible to safely model it as a uniform access accelerator. Such a model can be used to assign the ABU budget and compute safe upper bounds on the response time of the Sobel HW-task. The case study application has been tested with a set of four experiments considering different HW-task periods and ABU budgets. Table 5.5 summarizes the parameters used for the experiments. The ABU period P is set to 128 clock cycles in all of the experiments. The results are reported in Figure 5.10, which compares the response times calculated using the proposed response-time analysis, plotted as solid bars, with the longest-observed response times obtained on the real hardware, illustrated with striped bars. Measurements on the hardware have been performed as described in the previous section.

The experimental results show that the ABU is indeed effective even considering a case-study application comprising a realistic hardware workload suited for signal and image processing. The response times bounds obtained with the analysis are close to the longest-observed values with a maximum relative error of 3% in the case of HW-tasks with uniform demand. As expected, the maximum difference between the bound and the measurements (13%) occurs for the Sobel HW-task, since it has been pessimistically modeled by assuming a continuous bus access at its maximum rate.

Table 5.5: Configuration parameters for the case study (HW-task periods and ABU budgets).

Task	Experiment							
	1		2		3		4	
	T_i [ms]	B_i [tr]	T_i [ms]	B_i [tr]	T_i [ms]	B_i [tr]	T_i [ms]	B_i [tr]
FIR	6	176	6	160	8	96	6	160
Sobel	7	160	8	144	9	112	12	96
DMA-2	10	80	12	64	6	144	7	128
DMA-1	12	64	7	112	7	128	10	80

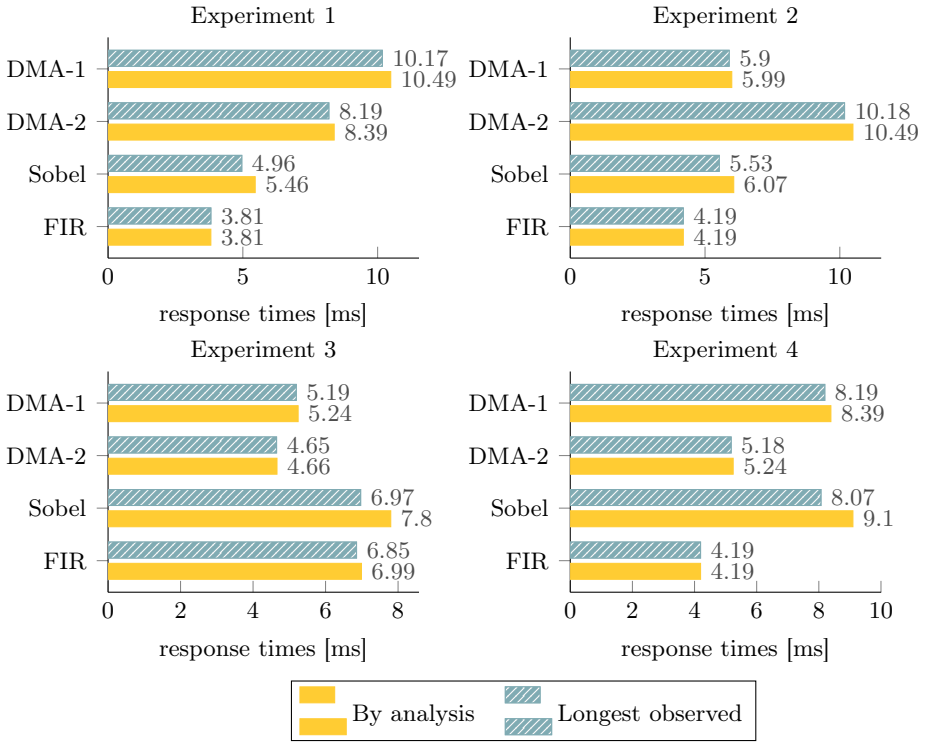

Figure 5.10: Response times for the case study.

Chapter 6

Related work

This chapter compares the work presented in this thesis with other approaches available in the literature. First, the FRED framework is compared in detail with existing solutions by systematically classifying the differences within a taxonomy. Then, current solutions for enhancing bus predictability and safety are compared with the proposed ABU highlighing the key differences.

6.1 Predictable hardware acceleration on FPGA

The solutions proposed in the literature to exploit FPGA-based hardware acceleration are quite heterogeneous due to the evolution of such platforms and the wide range of applications that can take advantage of this technology. The intrinsic parallelism, the reduced interference among the running activities, and the reduced variability in the execution made such a technology appealing for real-time applications, ranging from network management [44] to scheduling of hard [48] and soft [28] tasks. However, current solutions are limited to static or slowly evolving scenarios. Before analyzing the related work on DPR for real-time task scheduling, a taxonomy is first introduced to classify the existing solutions and precisely position the proposed approach with respect to the literature.

6.1.1 Taxonomy

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

Reconfiguration approaches

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

Allocation methods

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

reconfiguration times due to the instantiation of communication channels and the increased traffic on the FRI due to the additional data transfer.

FRI model

The FRI plays a central role in FPGAs with DPR, thus, building a proper model of the FRI is crucial for estimating worst-case delays and enabling a real-time analysis. The easiest approach is to reduce complexity by considering reconfiguration delays negligible. This is a strong unrealistic assumption, considering that, in current FPGAs, reconfiguration delays can have the same order of magnitude of task execution times. A simple approximation can be obtained using a constant reconfiguration time. However, since the reconfiguration time is proportional to the number of logic cells to be reconfigured, and the FRI is a shared resource, providing a safe bound would introduce a huge pessimism in the analysis. Less pessimistic values can be obtained considering the reconfiguration time composed by two elements: one proportional to the number of logic cells to be reconfigured and one due to the time spent in waiting for the FRI. Most of the works focused on kernel mechanisms considered an FRI model tailored to real solutions, as the Xilinx ICAP and PCAP ports [50].

Task model

Modern heterogeneous platforms include general purpose-processors tightly coupled with FPGA fabric on the same chip [77]. On such platforms it is thus possible to execute both HW-tasks, running on the FGPA, and software tasks (SW-tasks), running on the processors.

Related work analysis

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

Di Natale and Bini [48] proposed an optimization method to partition the reconfigurable area of a homogeneous FPGA platform into slots to be allocated to HW-tasks and softcores running the remaining tasks. Given the high computational complexity of the method, this approach can only be used off-line to obtain a static task allocation, hence it does not exploit the advantages of the dynamic reconfiguration. Pellizzoni and Caccamo [53] addressed a similar problem in a more dynamic scenario, proposing an allocation scheme and an admission test to provide real-time guarantees of applications supporting mode changes, where tasks can either be executed in software on a CPU or in hardware on the FPGA.

Danne and Platzner [21] presented two algorithms (one EDF-based and one server-based) to schedule only preemptive HW-tasks, but the model

adopted for the FPGA platform is quite simple and does not consider any reconfiguration time and allocation constraints. Saha et. al. [59] presented a new scheduling algorithm for preemptable HW-tasks, exploiting the higher speed and the improved capabilities of modern reconfiguration interfaces to dynamically change the allocation every time a task terminates. However, this approach assumes a homogeneous partition and a fixed reconfiguration time, which can lead to a huge waste of the area and a high pessimism in the analysis. In summary, in all the works cited above, the models used for the FPGA and the reconfiguration interface are too simple to describe the limitations of the available platforms, and the corresponding approaches do not fully exploit reconfiguration capabilities under real-time constraints.

Dittmann and Frank [24] addressed the analysis of reconfiguration requests as a single core scheduling problem. The paper assumes a single set of homogeneous slots managed by a non-preemptable FRI and considers only HW-tasks (SW-tasks are not taken into account). Unfortunately, due to missing proofs, it is not clear how response-time bounds follow. In addition, the authors did not investigate sustainability issues and their analysis may be affected by later-discovered misconceptions concerning non-preemptive fixed-priority scheduling [23].

Other authors proposed methods for supporting a job-level reconfiguration from a system perspective. The easiest solution is to communicate with a HW-task through proper software stubs that interact with the kernel scheduler and manage the HW-tasks at the application level. Another approach is to extend the operating system to provide specific primitives for scheduling, allocating, and programming HW-tasks, along with those related to SW-tasks management. The second method increases the complexity of the kernel, but reduces the issues related to the effects that two non-interacting scheduling levels have on the pessimism in the timing analysis. The approach based on the use of SW stubs has been followed by most of the authors presenting the few working solutions actually available, because it requires less modification in the kernel and has a reduced effects of the average performance, which is the main concern in these solutions. For instance, Lübbers and Platzner [42] proposed the ReconOS operating system, which extends the classic multi-threading programming model to hardware activities executed on a reconfigurable device. HW-tasks interact with SW-tasks threads trough a custom developed POSIX-style API, using the same operating system mechanisms, like semaphores, condition variables, and message queues. Originally designed for fully reconfigurable FPGAs, this solution has then been extended by the same authors to support partial reconfiguration [41], with a cooperative multitasking approach to deal with slot contentions. More recently, Happe et. al. [30] proposed an extension to the ReconOS execution environment to provide HW-tasks preemptability. However, its focus is on hardware enabling technologies, not on a kernel support for exploiting this capability. Iturbe et al. [33] presented the R3TOS operating system

to support a more dynamic task allocation, exploiting the reconfiguration interface to avoid static communication channels. In their solution, scheduling and allocation of HW-task is performed by a module, called HWuK, which is also in charge of controlling the programming interface in an exclusive manner. The authors proposed a HW-task model and algorithms for their scheduling and allocation. However, a worst-case analysis is not provided and nothing is said on the schedulability of SW-tasks.

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

6.1.2 Classification

Table 6.1 classifies the presented works according to the proposed taxonomy, also highlighting the availability of a real-time analysis (RTA) to better emphasize the differences with respect to the proposed approach. Summarizing, different approaches have been proposed to exploit the advantages of DPR-enabled FPGAs, but none of them provided worst-case bounds for enabling a worst-case timing analysis of real-time sets of mixed HW-tasks and SW-tasks. In addition, most of the previous work did not consider heterogeneous FPGA slots. To overcome these limitations, FRED uses a heterogeneous slotted-based design to make reconfiguration times more predictable and derive a schedulability analysis for real-time applications exploiting DPR capabilities. FPGA reconfiguration is managed at the job level and the schedulability analysis takes into account the delays and the constraints coming from the FRI.

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

Table 6.1: Classification of the related work.

6.1.3 Linux support

Concerning the support available on Linux for FPGA-based hardware acceleration and partial reconfiguration, there is a growing interest in providing system support to these features. However, the current mainline kernel only provides baseline support for partial reconfiguration, in the form of a common abstraction layer above the different vendor-specific drivers. Brodersen et al. address this problem by proposing BORPH [65], which is an extension of Linux to allow the co-scheduling of software tasks and hardware tasks. Similar to the ReconOS solution, they provide inter-task communication using standard UNIX interprocess communication (IPC) semantics. Unfortunately, the project is no more active. Hence the last release is based on an old version of the Linux kernel, and the support for modern FPGA platforms is missing.

6.2 Enhancing bus predictability

Resource reservation techniques have been introduced in the context of real-time systems for CPUs scheduling [56, 2, 11] and applied to share other computational resources like programmable GPUs [36, 35]. Essentially, the idea is assigning to each entity (e.g., task) a fraction of a shared resource under contention (e.g., processor) in order to provide temporal isolation. Similarly, this work adjusts the same approach to the contention of the AMBA AXI bus in the context of hardware-programmable SoC FPGA platforms.

Many research efforts have been dedicated to the problem of bus contention in real-time systems. Schliecker et al. [61] use an event-based model to estimate delays for communications and computation activities on a multicore SoC platform. Pellizzoni and Caccamo [52] analyzed the interaction between CPU and peripherals while contending a shared main memory within a theoretical framework and proposed a conceptual solution based on a hardware server to control the unpredictable behavior of COTS peripherals. Betti et al. [10] presented a framework for providing real-time guarantees in a COTS platform. Each peripheral within the platform is supervised by a "real-time bridge" controlled by a system-wide peripheral scheduler. Their framework has been developed and evaluated on PC platforms with PCI Express bus while our approach considers on-chip buses for integrated SoC-FPGA platforms.

In the context of memory contention on multicore platforms, Agrawal et al. [4] presented a technique to perform the analysis both WCETs and schedulability of real-time activities under dynamic memory scheduling. Yum et al. [75] proposed a memory bandwidth reservation mechanism named MemGuard. The system provides memory performance isolation employing a bandwidth regulator for each core. The bandwidth regulators enforce a budgeting mechanism and are implemented using performance counters. Our approach is somehow related to this work since both consider bandwidth regulation of bus master agents. However, while MemGuard considers inter-

core interference on an Intel chip multiprocessor, our work considers bus interference generated by hardware accelerators on the AMBA AXI bus.

In the domain of packet switching networks, many efforts have been dedicated to the modeling and the analysis of traffic scheduling algorithms to provide quality of service (QoS) guarantees [20, 69]. Such methodologies have also been employed on SoCs platforms to develop and analyze arbiters for heavily-contented resources like the system memory [5, 27]. The ABU can be improved by leveraging the results of these works. Concerning the development of on-chip communication infrastructures for SoC platforms, transaction-based buses and packet-based networks on chip (NoC) remain the dominant approaches [57]. Typically, arbitration for on-chip interconnects is performed using Fixed Priority, Round Robin, and Time-Division Multiple Access (TDMA). Poletti et al. presented a performance analysis comparing different arbitration policies for SoCs platforms in [55]. A TDMA-based arbitration scheme with dynamic timeslot allocation is employed in [57, 14] to improve system predictability while providing good average-case performance. Lahiri et al. [39] proposed a statistical approach to arbitration using a ticket-based random selection which was further extended by other works [17, 40] to improve predictability. Steine et al. [68] proposed a TDMA budget based scheduler for data flow applications, which has been used by Staschulat et al. [67] for memory arbitration. However, while the latter work is explicitly targeted at embedded systems, it is still limited to dataflow applications. Bourgade [13] proposed a bus arbitration scheme for multicore platforms designed to ease the estimation of the tasks' worst-case execution times. Reconfigurable bus arbiters [74, 66] can be dynamically configured to change the arbitration policy depending on the application requirements. Likewise, several papers in the literature addressed the problem of designing predictable memory controllers for multi-core architectures. Guo et al. [29] presented a comparative analysis of predictable DRAM controllers.

Conclusions

This thesis addressed the problem of enhancing system predictability while using FPGA-based hardware acceleration for real-time systems. The first part of this thesis presented FRED, a framework designed for supporting FPGA hardware acceleration and fabric resources "virtualization" through dynamic partial reconfiguration. The FRED framework has been convinced to be predictable by design, ensuring bounded response times for software activities that make use of hardware acceleration for speeding up their computations. The proposed framework is based on a platform model that matches the capabilities and limitations of modern SoC-FPGA platforms. After presenting the model of the platform and the computational activities, a scheduling infrastructure has been proposed. Such a scheduling infrastructure has been designed to bound the delays experienced by the software activities that make use of hardware acceleration. FRED has been prototyped on FreeRTOS and then fully implemented on a rich operating system such as GNU/Linux. The Linux implementation has been evaluated with a realistic case study application showing that practical applications can indeed benefit from dynamic hardware acceleration on SoC-FPGA platforms. The second part of this thesis presented the ABU, a mechanism for enhancing bus predictability and system safety on SoC-FPGA platforms. The ABU provides a hardware-based reservation mechanism for the AMBA AXI bus aimed at isolating hardware accelerators. After describing the internal architecture of the ABU, a response-time in the bandwidth domain has been presented to verify the schedulability of a set of hardware accelerators under real-time constraints. The proposed mechanism has been implemented and validated with real-word hardware accelerators to demonstrate its practical applicability. A substantial experimental evaluation confirmed the effectiveness of the proposed solution, showing that it can be implemented with a limited amount of FPGA logic resources.

Future research efforts will be dedicated to improving the integration of the proposed mechanisms and to support newer SoC-FPGA platforms. Moreover, another promising direction for further research is the integration of the proposed mechanisms into a hypervisor. In the domain of CPS, the isolation and reservation support provided by hypervisors are becoming crucial for safely integrating several functionalities with different criticality on the same platform. Applying the proposed methodologies for FPGA-based hardware acceleration in this domain can provide substantial performance improvements while guaranteeing the necessary predictability.

List of publications

This thesis is based on the following publications:

- Marco Pagani, Enrico Rossi, Alessandro Biondi, Mauro Marinoni, Giuseppe Lipari, and Giorgio Buttazzo, "A Bandwidth Reservation Mechanism for AXI-based Hardware Accelerators on FPGAs", *In Proceedings of the 31th Euromicro Conference on Real-Time Systems (ECRTS 19)*, Stuttgart, Germany, July 9-12, 2019.
- Marco Pagani, Alessio Balsini, Alessandro Biondi, Mauro Marinoni and Giorgio Buttazzo, "A Linux-based Support for Developing Real-Time Applications on Heterogeneous Platforms with Dynamic FPGA Reconfiguration", *In Proceedings of the 30th IEEE International System-on-Chip Conference (SOCC 2017)*, Munich, Germany, September 5-8, 2017.
- Alessandro Biondi, Alessio Balsini, Marco Pagani, Enrico Rossi, Mauro Marinoni and Giorgio Buttazzo, "A Framework for Supporting Real-Time Applications on Dynamic Reconfigurable FPGAs", *In Proceedings of the 37th IEEE Real-Time Systems Symposium (RTSS 16)*, Porto, Portugal, November 29 - December 2, 2016.

The Ph.D. program also included other research efforts which produced the following publications:

- Francesco Restuccia, Marco Pagani, Alessandro Biondi, Mauro Marinoni, and Giorgio Buttazzo, "Modeling and Analysis of Bus Contention for Hardware Accelerators on FPGA SoCs ", *To be presented at the the 32th Euromicro Conference on Real-Time Systems (ECRTS 20).*
- Francesco Restuccia, Marco Pagani, Alessandro Biondi, Mauro Marinoni, and Giorgio Buttazzo, "Is Your Bus Arbiter Really Fair? Restoring Fairness in AXI Interconnects for FPGA SoCs", ACM Transactions on Embedded Computing Systems (TECS), 2019, 18.5s: 51. *Presented at*

the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES 2019), New York, USA, October 13 - 18, 2019.

Acknowledgements

The work presented in this thesis would not have been possible without the support of several people to whom I am sincerely grateful. I would like to thanks my supervisors Giorgio Buttazzo and Giuseppe Lipari, for their guidance and for supporting the development of this thesis. Following their advice, I learned many things in these years, not only in the form of scientific and technical knowledge but also in terms of communication and work organization skills. I also would like to say a big thanks to my tutor Mauro Marinoni whose constant support has been crucial in these years, from my master's program to the development of this thesis. Moreover, I'm deeply grateful to Alessandro Biondi, whose continuous support has been fundamental for the development of this work. Indeed, his scientific advice and his thought-provoking comments have been a staple for the development of this thesis.

Furthermore, I'm sincerely grateful to all fellow lab mates of the ReTiS lab who supported me during these years. In particular, Francesco Restuccia and Biruk Seyoum for the work that we have done together and for being excellent teammates. Daniel Casini, Paolo Pazzaglia, Daniel Bristot de Oliveira, and Caroline Brandberg for being great course mates always willing to help during the moments of discouragement. Alessio Balsini, Luigi Pannocchi, Fabrizio Gambini, Carmelo di Franco, Enrico Rossi, and all the other guys of the old guard for the good times we spent together. Moreover, I sincerely thank Pasquale Buonocunto for encouraging me to continue with the Ph.D. after the conclusion of my master's program. Without his support, I probably would have lost the opportunity of doing this thesis. Likewise, I would like to thank Antonio Prete for encouraging me to continue my education, after my bachelor's degree, with a master's degree in Embedded Computing Systems. In hindsight, it has been an excellent decision.

I also would like to thank all the people I met at the IRCICA during the year I spent in Lille. In particular, Houssam Eddine Zahaf, Pierre Falez, and Frédéric Fort for being great office mates and for helping me in many situations. Jordy Ruiz for the interesting and engaging discussions we had.

Julien Forget, Philippe Devienne, and all the IRCICA faculty staff for being always available for helping me during my stay. I can definitively say that the period I spent in Lille has been an excellent work and life experience.

Finally, I am profoundly grateful to my family and to all my friends outside the research world for their patience and presence during these demanding years. This work would not have been possible without their unconditional support.

Bibliography

[1] *7 Xeries FPGAs Configurable Logic Block*. UG474. Xilinx (cit. on p. 23).

[2] Luca Abeni and Giorgio Buttazzo. "Integrating multimedia applications in hard real-time systems". In: *Real-Time Systems Symposium, 1998. Proceedings. The 19th IEEE*. IEEE. 1998, pp. 4–13 (cit. on p. 99).

[3] *Accelerating OpenCV Applications with Zynq-7000 All Programmable SoC using Vivado HLS Video Libraries*. XAPP1167. Rev. 3.0. Xilinx. June 2015 (cit. on p. 59).

[4] Ankit Agrawal et al. "Analysis of dynamic memory bandwidth regulation in multi-core real-time systems". In: *2018 IEEE Real-Time Systems Symposium (RTSS)*. IEEE. 2018, pp. 230–241 (cit. on p. 99).

[5] Benny Akesson, Liesbeth Steffens, and Kees Goossens. "Efficient service allocation in hardware using credit-controlled static-priority arbitration". In: *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE. 2009, pp. 59–68 (cit. on p. 100).

[6] *AMBA AXI and ACE Protocol Specification*. ARM. 2011 (cit. on pp. 71, 72).

[7] James H. Anderson, Philip Holman, and Anand Srinivasan. "Fair Scheduling of Real-Time Tasks on Multiprocessors". In: *Handbook of Scheduling - Algorithms, Models, and Performance Analysis*. Chapman and Hall/CRC, 2004 (cit. on p. 78).

[8] *AXI Interconnect, LogiCORE IP Product Guide*. PG059. Xilinx Inc. 2018 (cit. on pp. 48, 71, 72).

[9] C. Beckhoff, D. Koch, and J. Torresen. "Go Ahead: A Partial Reconfiguration Framework". In: *Proc. of the 20th Annual IEEE Int. Symposium on Field-Programmable Custom Computing Machines*. Toronto, Canada, Apr. 2012 (cit. on p. 30).

[10] E. Betti et al. “Real-Time I/O Management System with COTS Peripherals”. In: *IEEE Transactions on Computers* 62.1 (Jan. 2013), pp. 45–58. ISSN: 0018-9340. DOI: 10.1109/TC.2011.202 (cit. on p. 99).

[11] Alessandro Biondi, Alessandra Melani, and Marko Bertogna. “Hard constant bandwidth server: Comprehensive formulation and critical scenarios”. In: *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*. IEEE. 2014, pp. 29–37 (cit. on p. 99).

[12] Alessandro Biondi et al. “A Framework for Supporting Real-Time Applications on Dynamic Reconfigurable FPGAs”. In: *Real-Time Systems Symposium (RTSS)*. 2016, pp. 1–12 (cit. on pp. 38, 39).

[13] Roman Bourgade, Christine Rochange, and Pascal Sainrat. “Predictable bus arbitration schemes for heterogeneous time-critical workloads running on multicore processors”. In: *Emerging Technologies & Factory Automation (ETFA), 2011 IEEE 16th Conference on*. IEEE. 2011, pp. 1–4 (cit. on p. 100).

[14] Paolo Burgio et al. “Adaptive TDMA bus allocation and elastic scheduling: A unified approach for enhancing robustness in multi-core RT systems”. In: *Computer Design (ICCD), 2010 IEEE International Conference on*. IEEE. 2010, pp. 187–194 (cit. on p. 100).

[15] Giorgio C Buttazzo. “Hartik: A real-time kernel for robotics applications”. In: *Real-Time Systems Symposium, 1993., Proceedings*. IEEE. 1993, pp. 201–205 (cit. on p. 60).

[16] Andrew Canis et al. “From software to accelerators with legup high-level synthesis”. In: *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. IEEE Press. 2013, p. 18 (cit. on p. 69).

[17] Chien-Hua Chen et al. “A real-time and bandwidth guaranteed arbitration algorithm for SoC bus communication”. In: *Design Automation, 2006. Asia and South Pacific Conference on*. IEEE. 2006, 6–pp (cit. on p. 100).

[18] *Convolutional Encoder, LogiCORE IP Product Guide*. PG026. Xilinx Inc. 2018 (cit. on p. 88).

[19] Ben Cope et al. “Performance comparison of graphics processors to reconfigurable logic: A case study”. In: *IEEE Transactions on computers* 59.4 (2010), pp. 433–448 (cit. on p. 17).

[20] Rene L Cruz et al. “A calculus for network delay, part I: Network elements in isolation”. In: *IEEE Transactions on information theory* 37.1 (1991), pp. 114–131 (cit. on p. 100).

[21] K. Danne and M. Platzner. “Periodic real-time scheduling for FPGA computers”. In: *Third International Workshop on Intelligent Solutions in Embedded System, 2005*. Hamburg, Germany, May 2005, pp. 117–127 (cit. on p. 96).

[22] Robert I. Davis and Alan Burns. “A Survey of Hard Real-time Scheduling for Multiprocessor Systems”. In: *ACM Comput. Surv.* 43.4 (2011) (cit. on p. 80).

[23] Robert I. Davis et al. “Controller Area Network (CAN) Schedulability Analysis: Refuted, Revisited and Revised”. In: *Real-Time System* 35.3 (2007), pp. 239–272 (cit. on p. 97).

[24] Florian Dittmann and Stefan Frank. “Hard Real-time Reconfiguration Port Scheduling”. In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. Nice, France, Apr. 2007 (cit. on pp. 33, 97).

[25] *Fast Fourier Transform, LogiCORE IP Product Guide*. PG109. Xilinx Inc. 2018 (cit. on p. 88).

[26] *FIR Compiler, LogiCORE IP Product Guide*. PG149. Xilinx Inc. 2018 (cit. on p. 88).

[27] Manil Dev Gomony et al. “A globally arbitrated memory tree for mixed-time-criticality systems”. In: *IEEE Transactions on Computers* 66.2 (2017), pp. 212–225 (cit. on p. 100).

[28] Ian Gray et al. “Transparent hardware synthesis of Java for predictable large-scale distributed systems”. In: *Proceedings of the Second International Workshop on FPGAs for Software Programmers (FSP)*. London, UK, Sept. 2015 (cit. on p. 95).

[29] Danlu Guo et al. “A comparative study of predictable dram controllers”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 17.2 (2018), p. 53 (cit. on p. 100).

[30] M. Happe, A. Traber, and A. Keller. “Preemptive hardware multitasking in ReconOS”. In: *Proceedings of the 11th International Symposium on Applied Reconfigurable Computing (ARC)*. Bochum, Germany, Apr. 2015, pp. 79–90 (cit. on p. 97).

[31] Dominik Honegger, Helen Oleynikova, and Marc Pollefeys. “Real-time and low latency embedded computer vision hardware based on a combination of fpga and mobile cpu”. In: *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*. IEEE. 2014, pp. 4930–4935 (cit. on p. 17).

[32] *Intel Stratix 10 Logic Array Blocksand Adaptive Logic Modules User-Guide*. UG-S10LAB. Intel (cit. on p. 23).

[33] Xabier Iturbe et al. “Microkernel Architecture and Hardware Abstraction Layer of a Reliable Reconfigurable Real-Time Operating System (R3TOS)”. In: *ACM Transactions on Reconfigurable Technology and Systems* 8.1 (Feb. 2015) (cit. on p. 97).

[34] Jan Moritz Joseph et al. “Design space exploration for a hardware-accelerated embedded real-time pose estimation using vivado HLS”. In: *ReConFigurable Computing and FPGAs (ReConFig), 2017 International Conference on*. IEEE. 2017, pp. 1–8 (cit. on p. 17).

[35] Shinpei Kato et al. “Resource sharing in GPU-accelerated windowing systems”. In: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*. IEEE. 2011, pp. 191–200 (cit. on p. 99).

[36] Shinpei Kato et al. “TimeGraph: GPU scheduling for real-time multi-tasking environments”. In: *Proc. USENIX ATC*. 2011, pp. 17–30 (cit. on p. 99).

[37] Dirk Koch. *Partial Reconfiguration on FPGAs: Architectures, Tools and Applications*. Springer-Verlag New York, Feb. 2012 (cit. on pp. 26, 30, 40).

[38] Ian Kuon and Jonathan Rose. “Measuring the gap between FPGAs and ASICs”. In: *IEEE Transactions on computer-aided design of integrated circuits and systems* 26.2 (2007), pp. 203–215 (cit. on p. 16).

[39] Kanishka Lahiri, Anand Raghunathan, and Ganesh Lakshminarayana. “The LOTTERYBUS on-chip communication architecture”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 14.6 (2006), pp. 596–608 (cit. on p. 100).

[40] Bu-Ching Lin et al. “A precise bandwidth control arbitration algorithm for hard real-time SoC buses”. In: *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*. IEEE Computer Society. 2007, pp. 165–170 (cit. on p. 100).

[41] Enno Lübbers and Marco Platzner. “Cooperative multithreading in dynamically reconfigurable systems.” In: *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*. Prague, Czech Republic, Aug. 2009 (cit. on p. 97).

[42] Enno Lübbers and Marco Platzner. “ReconOS: Multithreaded Programming for Reconfigurable Computers”. In: *ACM Transactions on Embedded Computing Systems* 9.1 (Oct. 2009), 8:1–8:33 (cit. on p. 97).

[43] Robert C Martin. “Design principles and design patterns”. In: *Object Mentor* 1.34 (2000), p. 597 (cit. on p. 53).

[44] Ernesto Martins, Luis Almeida, and José Alberto Fonseca. "An FPGA-based Coprocessor for Real-time Fieldbus Traffic Scheduling: Architecture and Implementation". In: *Journal of Systems Architecture* 51.1 (2005), pp. 29–44 (cit. on p. 95).

[45] J. Y. Mignolet et al. "Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip". In: *Proceedings of DATE*. Munich, Germany, Mar. 2003 (cit. on p. 33).

[46] Razvan Nane et al. "A survey and evaluation of FPGA high-level synthesis tools". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.10 (2015), pp. 1591–1604 (cit. on p. 13).

[47] Razvan Nane et al. "A survey and evaluation of fpga high-level synthesis tools". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.10 (2016), pp. 1591–1604 (cit. on p. 69).

[48] M. Di Natale and E. Bini. "Optimizing the FPGA Implementation of HRT Systems". In: *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS)*. Bellevue, WA, USA, Apr. 2007 (cit. on pp. 95, 96).

[49] G. Nelissen et al. "Timing Analysis of Fixed Priority Self-Suspending Sporadic Tasks". In: *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*. Lund, Sweden, July 2015 (cit. on p. 38).

[50] *Partial Reconfiguration User Guide*. UG702. v14.1. Xilinx. 2012 (cit. on pp. 30, 96).

[51] Karl Pauwels et al. "A comparison of FPGA and GPU for real-time phase-based optical flow, stereo, and local image features". In: *IEEE Transactions on Computers* 61.7 (2012), pp. 999–1012 (cit. on p. 17).

[52] R. Pellizzoni and M. Caccamo. "Impact of Peripheral-Processor Interference on WCET Analysis of Real-Time Embedded Systems". In: *IEEE Transactions on Computers* 59.3 (Mar. 2010), pp. 400–415. ISSN: 0018-9340. DOI: 10.1109/TC.2009.156 (cit. on p. 99).

[53] R. Pellizzoni and M. Caccamo. "Real-Time Management of Hardware and Software Tasks for FPGA-based Embedded Systems". In: *IEEE Transactions on Computers* 56.12 (Dec. 2007), pp. 1666–1680 (cit. on p. 96).

[54] Adam Petersen. "Patterns in C". In: () (cit. on p. 53).

[55] Francesco Poletti et al. "Performance analysis of arbitration policies for SoC communication architectures". In: *Design Automation for Embedded Systems* 8.2-3 (2003), pp. 189–210 (cit. on p. 100).

[56] Ragunathan Rajkumar et al. “Resource kernels: A resource-centric approach to real-time and multimedia systems”. In: *Multimedia Computing and Networking 1998*. Vol. 3310. International Society for Optics and Photonics. 1997, pp. 150–165 (cit. on p. 99).

[57] Thomas D Richardson et al. “A hybrid SoC interconnect with dynamic TDMA-based transaction-less buses and on-chip networks”. In: *VLSI Design, 2006. Held jointly with 5th International Conference on Embedded Systems and Design., 19th International Conference on*. IEEE. 2006, 8–pp (cit. on p. 100).

[58] Mohammadsadegh Sadri et al. “Energy and performance exploration of accelerator coherency port using Xilinx ZYNQ”. In: *Proceedings of the 10th FPGAworld Conference*. ACM. 2013, p. 5 (cit. on p. 40).

[59] S. Saha, A. Sarkar, and A. Chakrabarti. “Scheduling Dynamic Hard Real-Time Task Sets on Fully and Partially Reconfigurable Platforms”. In: *IEEE Embedded Systems Letters* 7.1 (Mar. 2015), pp. 23–26 (cit. on p. 97).

[60] Miro Samek. *Object-Oriented Programming in C*. 2019 (cit. on p. 53).

[61] Simon Schliecker et al. “Reliable performance analysis of a multicore multithreaded system-on-chip”. In: *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*. ACM. 2008, pp. 161–166 (cit. on p. 99).

[62] Douglas C Schmidt. “Reactor: An object behavioral pattern for concurrent event demultiplexing and dispatching”. In: (1995) (cit. on p. 53).

[63] Axel-Tobias Schreiner. “Object oriented programming with ANSI-C”. In: (1993) (cit. on p. 53).

[64] *SmartConnect, LogiCORE IP Product Guide*. PG247. Xilinx Inc. 2018 (cit. on pp. 71, 72).

[65] Hayden Kwok-Hay So and Robert Brodersen. “A Unified Hardware/Software Runtime Environment for FPGA-based Reconfigurable Computers Using BORPH”. In: *ACM Transactions on Embedded Computing Systems* 7.2 (Jan. 2008), 14:1–14:28 (cit. on p. 99).

[66] Éricles Sousa et al. “Runtime reconfigurable bus arbitration for concurrent applications on heterogeneous MPSoC architectures”. In: *Digital System Design (DSD), 2014 17th Euromicro Conference on*. IEEE. 2014, pp. 74–81 (cit. on p. 100).

[67] Jan Staschulat and Marco Bekooij. “Dataflow models for shared memory access latency analysis”. In: *Proceedings of the seventh ACM international conference on Embedded software*. ACM. 2009, pp. 275–284 (cit. on p. 100).

[68] Marcel Steine, Marco Bekooij, and Maarten Wiggers. "A priority-based budget scheduler with conservative dataflow model". In: *Digital System Design, Architectures, Methods and Tools, 2009. DSD'09. 12th Euromicro Conference on*. IEEE. 2009, pp. 37–44 (cit. on p. 100).

[69] Dimitrios Stiliadis and Anujan Varma. "Latency-rate servers: a general model for analysis of traffic scheduling algorithms". In: *IEEE/ACM Transactions on networking* 6.5 (1998), pp. 611–624 (cit. on p. 100).

[70] *UltraScale Architecture Configurable Logic Block*. Xilinx (cit. on p. 23).

[71] *Using Quality of Service (QoS) Capabilities in Zynq-7000 AP SoC Devices*. XAPP1266. Xilinx Inc. July 2015 (cit. on p. 70).

[72] Stylianos I Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. "Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions". In: *ACM Computing Surveys (CSUR)* 51.3 (2018), p. 56 (cit. on p. 17).

[73] *Vivado Design Suite User Guide: Partial Reconfiguration*. UG909. Xilinx (cit. on pp. 26, 30, 40, 45–47).

[74] Ching-Chien Yuan et al. "A reconfigurable arbiter for SOC applications". In: *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on*. IEEE. 2008, pp. 713–716 (cit. on p. 100).

[75] H. Yun et al. "MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms". In: *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Apr. 2013, pp. 55–64 (cit. on p. 99).

[76] *Zynq UltraScale+ Device - Reference Manual*. UG1085. Xilinx (cit. on p. 45).

[77] *Zynq-7000 All Programmable SoC - Reference Manual*. UG585. Xilinx (cit. on pp. 30, 39, 42, 44, 45, 96).